



Yazılım Mühendisliğinde Yapay Zeka Paradigması: Teorik Temeller, Metodolojik Dönüşümler ve Epistemolojik Etkiler

Yazılım Mühendisliği ve Yapay Zeka Araştırmaları

Giriş: Ontolojik Bir Kırılma Noktasındayız

Yazılım mühendisliği disiplini, yapısal programlamadan nesne yönelimli tasarıma, şelale modelinden çevik yaklaşılara kadar sayısız metodolojik evrim geçirmiştir. Ancak yapay zekanın ve özellikle derin öğrenme tekniklerinin yazılım geliştirme yaşam döngüsüne entegrasyonu, sadece metodolojik bir yenilik değil, aynı zamanda **ontolojik bir kırılma** yaratmaktadır.

Deterministik kodlama paradigmalarından (Software 1.0) olasılıksal ve veriye dayalı optimizasyon paradigmalarına (Software 2.0) geçiş, disiplinin epistemolojik temellerini yeniden tanımlamayı zorunlu kılmaktadır. Geleneksel yazılım mühendisliğinde "doğruluk" (correctness), mantıksal kanıtlanabilirlik ve spesifikasyonlara tam uyum ile tanımlanırken, yeni paradigmada yazılım artık sadece inşa edilen değil, aynı zamanda optimize edilen ve veriden öğrenilen bir varlık haline gelmiştir.

Metodolojik Evrim Süreci

- Yapısal Programlama → Nesne Yönelimli Tasarım
- Şelale Modeli → Çevik Yaklaşımlar
- Deterministik Kodlama → Olasılıksal Optimizasyon
- Açık Talimatlar → Veri Odaklı Öğrenme

Bu dönüşüm, gereksinim mühendisliğinden mimari tasarıma, kod üretiminden test süreçlerine kadar her aşamada teorik modellerin yeniden ele alınmasını gerektirmektedir.

Sunum Yapısı ve Teorik Çerçeve

01

Teorik Temeller

Software 1.0'dan Software 2.0'a geçişin matematiği, türevlenebilir programlama ve AI4SE/SE4AI ayırmaları

02

Gereksinim Mühendisliği

LLM'lerin semantik devrimi, vektör uzayında gereksinimler ve prompt mühendisliği

03

Mimari ve Tasarım

Neural Architecture Search, mimari yeniden yapılandırma ve üretken tasarım sistemleri

04

Kod Üretimi

Transformer mimarisi, yapısal kod temsili ve nöro-semantik yaklaşımlar

05

Test ve Doğrulama

Oracle problemi, metamorfik test ve karar verilemezlik sınırları

06

Sosyo-Teknik Boyut

Bilişsel yük teorisi, teknoloji kabulü ve epistemolojik riskler

Software 1.0: Deterministik Paradigma

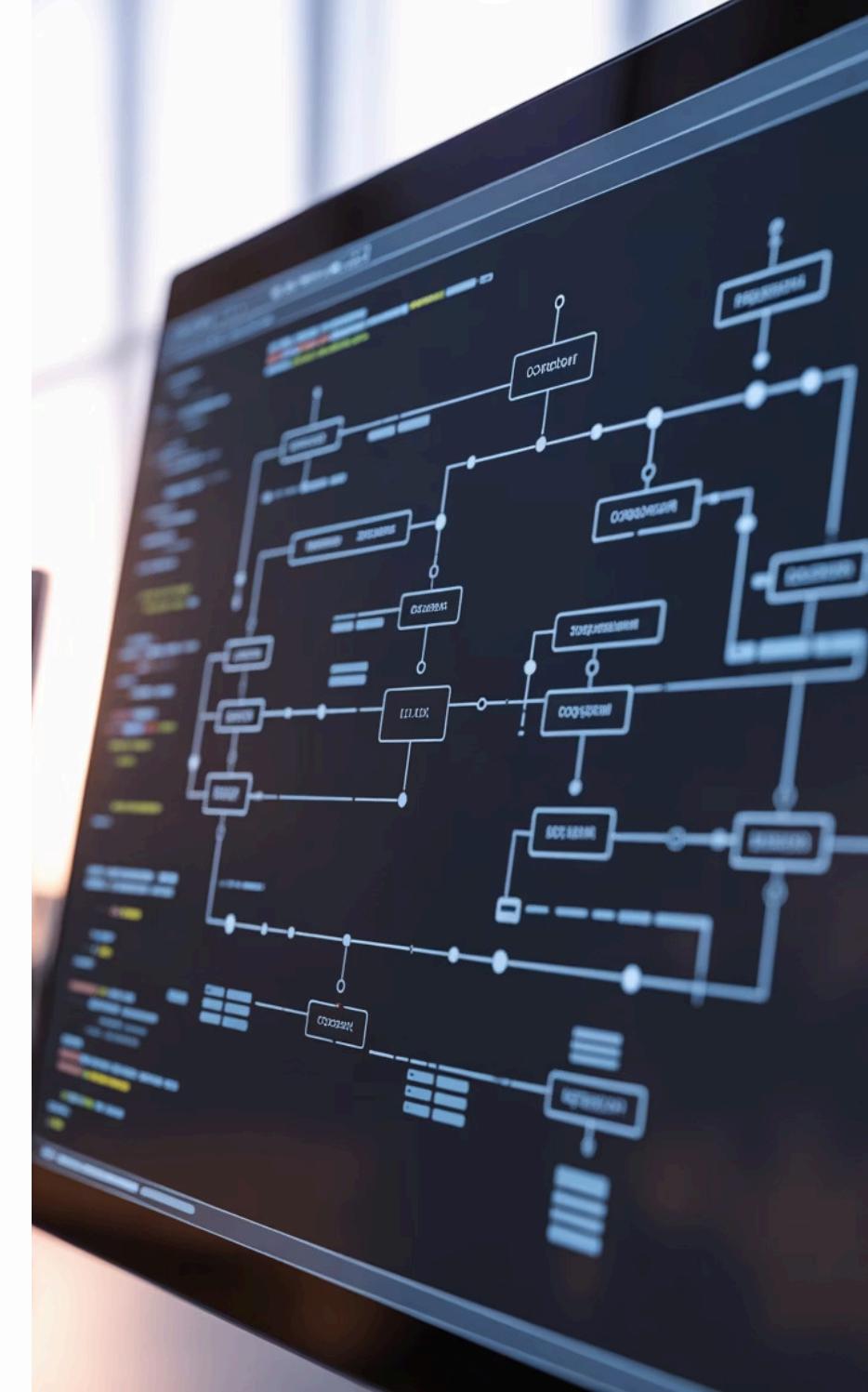
Geleneksel yazılım mühendisliğinin temel kabulü, kodun deterministik doğasıdır. Bir P programı ve x girdisi için, P(x) çıktısı her zaman aynıdır ve bu çıktı, programcının yazdığı mantıksal kuralların (R) bir sonucudur:

$$y = P(x) = R(x)$$

Bu paradigma, C++, Java, Python gibi dillerde yazılan açık talimat setlerini kapsar. Yazılım geliştirme süreci, insan zihninin problemleri ayrık mantık kurallarına bölmesi ve bunları bilgisayarın anlayabileceği açık talimatlara dönüştürmesi üzerine kuruludur.

Temel özellikler: Karmaşıklık arttıkça, insan programcının tüm köşe durumlarını (edge cases) öngörmesi ve kodlaması giderek imkansız hale gelir. Mantıksal kanıtlanabilirlik ve spesifikasyonlara tam uyum, doğruluğun temel ölçütleridir.

- Deterministik Kodun Özellikleri
 - Açık talimat setleri
 - Mantıksal kanıtlanabilirlik
 - Spesifikasyona tam uyum
 - İnsan tarafından yazılır ve okunur
 - Diskret matematik temelli



Software 2.0: Olasılıksal Paradigma

Software 2.0'da programın davranışları, açık kurallarla değil, bir optimizasyon süreci sonucunda belirlenen parametrelerle (θ) tanımlanır. Program, bir sinir ağı olarak modellenebilir:

$$y = f(x; \theta)$$

Yazılım geliştirme süreci artık kod yazmak değil, bir amaç fonksiyonunu (objective/loss function, L) minimize edecek θ parametrelerini bulmaktadır:

$$\theta^* = \arg \min_{\theta} \sum_i L(f(x_i; \theta), y_i)$$

Paradigma Kayması

Bu denklem, yazılım mühendisliğinin temel aktivitesini "algoritma tasarıımı"ndan "veri seti kürasyonu" ve "hedef fonksiyonu tasarıımı"na kaydırır. Geleneksel derleyicinin yerini, stokastik gradyan inişi (SGD) ve geri yayılım (backpropagation) algoritmaları alır.

Bu geçiş, yazılımın diskret matematik alanından **sürekli (continuous) matematik ve istatistik alanına** taşınması anlamına gelir. Yazılım artık sadece inşa edilen değil, optimize edilen ve öğrenilen bir varlıktır.

Yeni Geliştirme Döngüsü

- Veri seti toplama ve kürasyon
- Hedef fonksiyon tasarımı
- Model mimarisi seçimi
- Parametre optimizasyonu (training)
- Validasyon ve ince ayar
- Sürekli öğrenme ve adaptasyon

Türevlenebilir Programlama: Teorik Altyapı

Yann LeCun ve diğer teorisyenler tarafından önerilen "Türevlenebilir Programlama" (Differentiable Programming), Software 2.0 paradigmının teorik çatısını oluşturur. Bu yaklaşımın, modern derin öğrenme modelleri, parametreleri öğrenilebilir olan fonksiyonel blokların birleşimi olarak tanımlanır.

Bir yazılım sistemi, uçtan uca türevlenebilir bir hesaplama grafiği (computational graph) olarak ifade edilebiliyorsa, sistemin herhangi bir noktasındaki hatanın kaynağı, zincir kuralı (chain rule) kullanılarak hesaplanabilir ve otomatik olarak düzeltilebilir.

Geleneksel yazılımda bir if-else bloğu, gradyanın akışını kesen (non-differentiable) bir bariyerken, Software 2.0'da bu yapılar "yumuşak" (soft) dikkat mekanizmaları veya olasılıksal kapılar (probabilistic gates) ile değiştirilir. Bu sayede, yazılımın mantığı veri ile birlikte evrilebilir.

❑ Temel İlkeler

Kodun insan tarafından okunabilir olmasından ziyade, optimizasyon algoritmaları tarafından yönetilebilir olması gerekiği savunulmaktadır. Bu, yazılımın soyutlama seviyesini insan bilişsel sınırlarının ötesine taşıır.

AI4SE ve SE4AI: İki Yönlü Dönüşüm

AI4SE: AI for Software Engineering

Yapay zeka tekniklerinin (ML, NLP, Derin Öğrenme) yazılım geliştirme araçlarına entegre edilmesidir. Bu alan, mevcut yazılım mühendisliği pratiklerini AI ile iyileştirmeye odaklanır.

- Kod tamamlama ve üretimi
- Hata tahmini ve önleme
- Gereksinim analizi otomasyonu
- Test senaryosu üretimi
- Mimari optimizasyon

SE4AI: Software Engineering for AI

Yapay zeka içeren sistemlerin (örneğin otonom araçlar, öneri sistemleri) güvenilir, ölçülebilir ve sürdürülebilir bir şekilde inşa edilmesi için gerekli mühendislik disiplinidir.

- MLOps ve model yaşam döngüsü
- Veri yönetimi ve kalite kontrolü
- Model versiyonlama
- Sürekli doğrulama
- Dijital ikiz kavramları

Systems Engineering Research Center (SERC) yol haritaları, bu iki alanın giderek birbirine yakınsadığını göstermektedir. Özellikle otonom sistemlerde, sistemin davranışının operasyonel süreçte öğrenmeye devam ettiği için, geleneksel V-modeli gibi statik doğrulama yöntemleri yetersiz kalmaktadır.

Paradigma Karşılaştırması: Kritik Ayrımlar

Boyut	Software 1.0 (Deterministik)	Software 2.0 (Olasılıksal)
Temel Yapı	Açık mantıksal kurallar ve talimatlar	Optimize edilmiş parametre matrisleri
Geliştirme Süreci	Kod yazma ve algoritma tasarıımı	Veri kürasyon ve hedef fonksiyon tasarımı
Matematik Temeli	Diskret matematik, mantık	Sürekli matematik, olasılık, istatistik
Derleyici/Eğitici	Geleneksel derleyici	Stokastik gradyan inişi, backpropagation
Doğruluk Ölçütü	Mantıksal kanıtlanabilirlik	Validasyon metriklerinde performans
Hata Düzeltme	Kod revizyonu ve debugging	Yeniden eğitim ve hiperparametre ayarı
Okunabilirlik	İnsan için anlaşılabilir kod	Optimizasyon algoritmalarına yönelik
Ölçeklenme	Karmaşıklıkla zorlaşır	Veri ve hesaplama gücüyle ölçeklenir

Gelecek Vizyonu: Hibrit Yetkinlik Gerekliliği

Geleceğin yazılım mühendisi, hem AI algoritmalarını kullanan (AI4SE) hem de AI sistemlerini yöneten (SE4AI) hibrit bir yetkinlik setine sahip olmak zorundadır. Bu dönüşüm, disiplinin sınırlarını genişletirken, yeni sorumluluk alanları da yaratmaktadır.



Algoritmik Yetkinlik

Geleneksel veri yapıları, algoritma tasarımı ve kompleksite analizi bilgisi temel olmaya devam edecektir.



ML/DL Uzmanlığı

Makine öğrenimi ve derin öğrenme modellerinin teorik temelleri, mimari seçimleri ve optimizasyon teknikleri kritik öneme sahiptir.



Veri Mühendisliği

Veri toplama, temizleme, etiketleme ve kalite kontrolü, yazılım geliştirmenin merkezinde yer alacaktır.



Yeni Doğrulama Paradigmaları

Metamorfik test, olasılıksal doğrulama ve sürekli izleme teknikleri gereklidir.



Bölüm 1 Özeti: Ontolojik Kırılmanın Boyutları

1 Metodolojik Dönüşüm

Yazılım geliştirme, kod yazmaktan veri ve hedef fonksiyonu tasarlamaya evrilmiştir. Bu değişim, SDLC'nin her aşamasını kökten etkilemektedir.

3 Araç ve Tekniklerin Evrimi

Derleyicilerin yerini optimizasyon algoritmaları, açık kodun yerini öğrenilebilir parametreler almaktadır.

2 Matematiksel Temel Kayması

Diskret matematik ve mantıktan, sürekli matematik, olasılık ve istatistiğe geçiş, disiplinin epistemolojik zeminini yeniden tanımlamaktadır.

4 Yetkinlik Gereksinimlerinin Değişimi

AI4SE ve SE4AI kesişiminde, hibrit beceri setlerine sahip yeni nesil mühendislere ihtiyaç vardır.



Gereksinim Mühendisliğinde Semantik Devrim

Doğal Dil İşlemeden Büyük Dil Modellerine

Gereksinim Mühendisliği (Requirements Engineering - RE), yazılım projelerinin en belirsiz ve hataya açık aşamasıdır. Paydaşlardan alınan sözel veya yazılı isteklerin teknik spesifikasyonlara dönüştürülmesi, geleneksel olarak insan analistlerin yorumuna bağlıydı.

NLP tekniklerinin RE alanında kullanımı 1990'lara kadar uzansa da, 2023-2025 yılları arasındaki gelişmeler, "sınıflandırma" odaklı yaklaşımlardan "üretken" (generative) yaklaşımlara radikal bir geçiş işaret etmektedir. Eskiden NLP araçları, gereksinim metinlerindeki muğlaklıkları tespit etmek veya gereksinimleri "fonksiyonel/fonksiyonel olmayan" olarak sınıflandırmak için kullanılmıştı.

Ancak Transformer tabanlı Büyük Dil Modelleri (LLM), gereksinim elicitation (ortaya çıkarma), modelleme ve doğrulama süreçlerini kökten değiştirmektedir. LLM'ler, paydaş mülakatlarından otomatik olarak kullanıcı hikayeleri türetebilmekte, eksik senaryoları öngörebilmekte ve hatta gereksinimleri UML diyagramlarına veya formal spesifikasyon dillerine çevirebilmektedir.

Geleneksel NLP vs LLM Tabanlı RE: Paradigma Farkı

Özellik	Geleneksel NLP Yaklaşımı	LLM Tabanlı Generative RE
Temel Görev	Sınıflandırma, Varlık Çıkarımı	Üretim, Dönüştürme, Öztleme
Bağlam Anlayışı	Sınırlı (Cümle bazlı)	Geniş (Doküman/Proje bazlı)
Eğitim Verisi	Etiketli, alana özgü veri setleri	Genel amaçlı devasa korpuslar
Doğruluk Kaynağı	İnsan etiketlemesi	Olasılıksal tahmin (Halüsinsasyon riski)
Kullanıcı Etkileşimi	Statik analiz araçları	Sohbet tabanlı, interaktif (Prompting)
Semantik Derinlik	Yüzeysel, sözdizimsel	Derin semantik ilişkiler
Ölçeklenebilirlik	Alan spesifik, sınırlı	Çok alanlı, yüksek genelleme

Bu tablo, LLM'lerin gereksinim mühendisliğinde yarattığı köklü değişimi net bir şekilde ortaya koymaktadır. Ancak bu güç, beraberinde yeni riskler ve sorumluluklar da getirmektedir.

Vektör Uzayında Gereksinimler: Teorik Altyapı

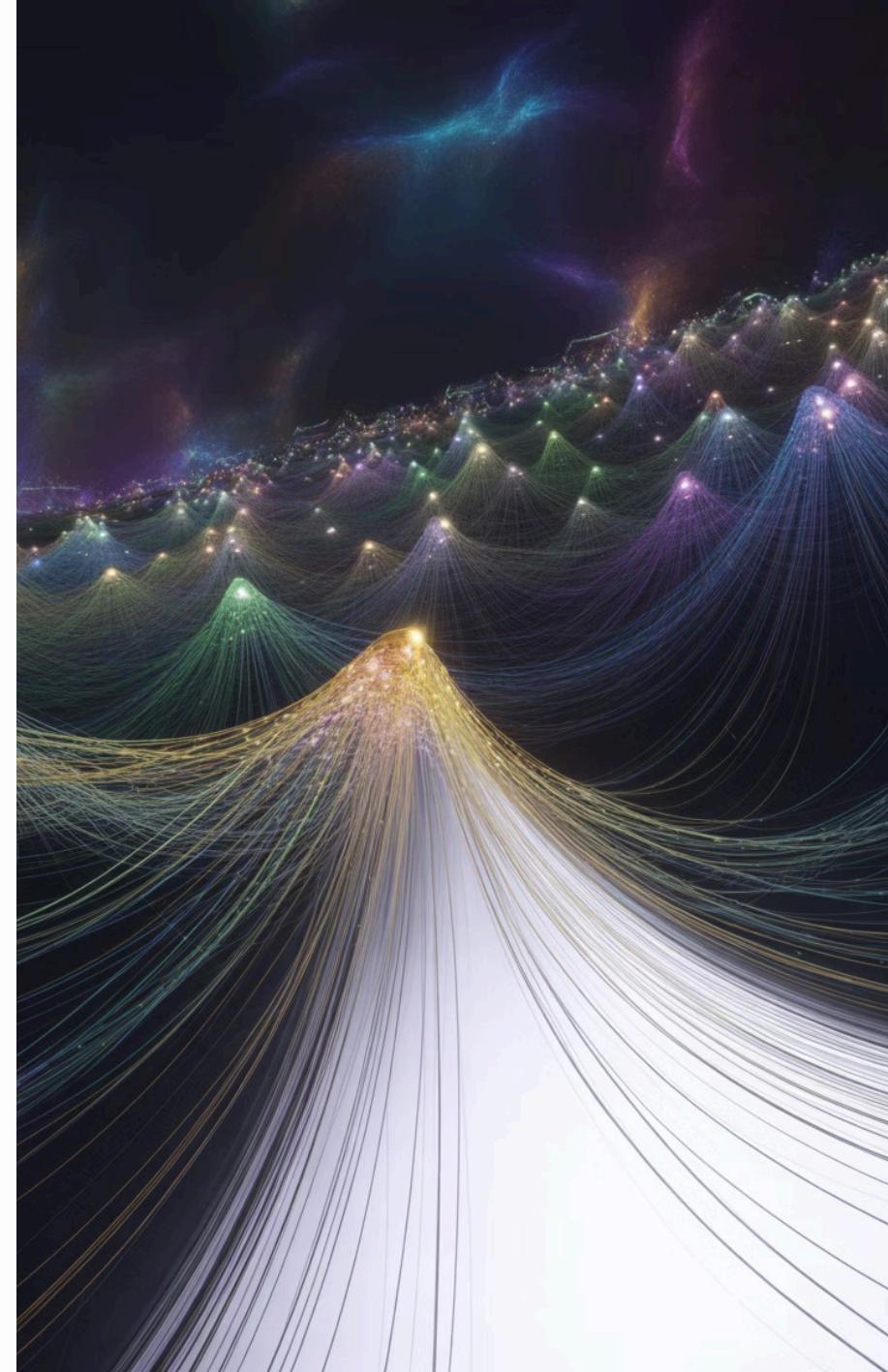
LLM'lerin RE alanındaki başarısının teorik temeli, gereksinimlerin **yüksek boyutlu vektör uzaylarında** (embedding space) temsil edilmesine dayanır. Geleneksel kelime eşleştirme yöntemlerinin aksine, vektör temsilleri "semantik yakınlığı" korur.

Örneğin, bir gereksinimdeki "hızlı yanıt süresi" ifadesi ile "düşük gecikme" (low latency) ifadesi, kelime olarak farklı olsalar da vektör uzayında birbirine yakındır. Bu matematiksel özellik, gereksinimlerin anlamsal ilişkilerinin hesaplanabilir hale gelmesini sağlar.

$$\text{similarity}(\textit{req}_1, \textit{req}_2) = \frac{\text{vec}(\textit{req}_1) \cdot \text{vec}(\textit{req}_2)}{\|\text{vec}(\textit{req}_1)\| \times \|\text{vec}(\textit{req}_2)\|}$$

Pratik Uygulamalar

- **Çelişki Tespiti:** Birbirile çelişen gereksinimler, vektör uzayında zıt yönlerde konumlanır
- **Tekrar Analizi:** Aynı anlamı taşıyan farklı ifadeler, kosinus benzerliği ile tespit edilebilir
- **Eksiklik Tanımlama:** Vektör kümelerindeki boşluklar, eksik senaryolara işaret edebilir
- **Önceliklendirme:** Benzer gereksinimlerin kümelenmesi, geliştirme önceliklerinin belirlenmesine yardımcı olur



Prompt Mühendisliği: Yeni Bir Yetkinlik Alanı

LLM'lerle etkili çalışmanın anahtarı, doğru "prompt" (komut/istem) tasarımda yatkınlıkta ve gereksinim analistleri için yeni bir yetkinlik alanı olarak "Prompt Mühendisliği"ni ortaya çıkarmıştır.



Zero-Shot Öğrenme

Model, hiçbir örnek verilmeden doğrudan görevi yerine getirir. Genel RE görevleri için yeterli olabilir.

Few-Shot Öğrenme

Modele 2-5 örnek verilerek, spesifik bir alan terminolojisi veya format öğretilir. Havacılık, finans gibi özel alanlar için kritiktir.

Chain-of-Thought

Model, cevaba adım adım ulaşması için yönlendirilir. Karmaşık gereksinim analizlerinde doğruluğu artırır.

Etkili Prompt Tasarımı İlkeleri

- Bağlam Sağlama:** Proje alanı, hedef kullanıcı profili ve teknik kısıtlar açıkça belirtilmelidir
- Rol Tanımlama:** LLM'e "Sen deneyimli bir gereksinim analistisin" gibi rol tanımları yapmak, çıktı kalitesini artırır
- Çıktı Formatı:** Beklenen çıktıının yapısı (UML, kullanıcı hikayesi, formal spesifikasiyon) net olmalıdır
- İteratif İyileştirme:** İlk çıktıının üzerine inşa eden, adım adım detaylandırma stratejisi kullanılmalıdır

Araştırma Bulgusu

Bağlam içi öğrenme (in-context learning) yeteneğinin, modellerin karmaşık RE görevlerinde, özel olarak eğitilmiş daha küçük modellerden daha başarılı olmasını sağladığı kanıtlanmıştır. Bu, genel amaçlı LLM'lerin, alan spesifik çözümlerden daha esnek ve güçlü olduğuna işaret eder.

Halüsinasyon Riski: Epistemolojik Bir Tehdit

LLM'lerin RE alanındaki en büyük teorik riski "halüsinasyon" (hallucination) problemidir. Üretken modeller, istatistiksel olarak olası ancak **olgusal olarak yanlış** veya **teknik olarak imkansız** gereksinimler üretebilir.

Halüsinasyon Türleri

1. Mantıksal Çelişki: Model, birbiriyile çelişen gereksinimler üretebilir. Örneğin, hem "sistem 7/24 çevrimdışı bakım yapabilmeli" hem de "sistem hiç duraksamadan çalışmalı" gibi.

2. Fiziksel İmkansızlık: Kullanılan donanımın kapasitesini aşan performans gereksinimleri önerilir. Örneğin, "sistem 1 nanosaniyede milyar kayıt taramalı" gibi.

3. Bağlamsal Uygunsuzluk: Proje bağlamına uymayan, alakasız özellikler eklenir. Örneğin, bir finans sisteme oyun mekaniklerinin dahil edilmesi.

Ripple Effect (Dalgalanma Etkisi)

Gereksinim aşamasında tespit edilemeyen hataların, yazılım yaşam döngüsünün ilerleyen aşamalarında katlanarak büyümesi, projenin başarısızlığına yol açabilir.

Halüsinasyonun maliyeti, keşif zamanına bağlı olarak üstel artar:

- Gereksinim aşamasında: 1x maliyet
- Tasarım aşamasında: 5-10x maliyet
- Uygulama aşamasında: 20-40x maliyet
- Test aşamasında: 50-100x maliyet
- Üretim sonrası: 100-200x maliyet

Human-in-the-Loop: Epistemik Otorite

1 LLM Üretimi

Büyük Dil Modeli, paydaş girdilerinden veya mevcut dokümantasyondan ilk gereksinim taslağını üretir.

2 İnsan İncelemesi

Deneyimli gereksinim analisti, üretilen içeriği alan bilgisi ve teknik uzmanlık ışığında değerlendirir.

3 Formal Doğrulama

Otomatik araçlar (kısıt çözücüler, statik analizörler) mantıksal tutarlılığı ve teknik uygulanabilirliği kontrol eder.

4 İteratif İyileştirme

Tespit edilen sorunlar LLM'e geri beslenir, model gereksinimleri düzeltir veya detaylandırır.

5 Paydaş Validasyonu

Son gereksinim seti, asıl paydaşlarla gözden geçirilir ve onaylanır.

Bu hibrit yaklaşım, AI'nın "yazar" rolünden ziyade "asistan" veya "fikir üretici" rolünde konumlandırılmasını sağlar. **Epistemik otorite insan mühendiste kalmalıdır.** LLM, hız ve kapsamlılık sağlarken, insan eleştirel düşünme, alan bilgisi ve etik sorumluluk getirir.

RE Otomasyonunun Pratik Uygulamaları



Kullanıcı Hikayesi Üretimi

LLM'ler, ham paydaş notlarından otomatik olarak "As a [user], I want [feature], so that [benefit]" formatında kullanıcı hikayeleri türetir.



UML Diyagram Sentezi

Metin tabanlı gereksinimlerden sınıf diyagramları, dizi diyagramları ve kullanım senaryosu diyagramları otomatik oluşturulur.



Uyumluluk Kontrolü

Gereksinimlerin yasal düzenlemelere (GDPR, HIPAA vb.) veya endüstri standartlarına (ISO, IEEE) uyumu otomatik değerlendirilir.



Çoklu Dil Desteği

Gereksinimlerin farklı dillere çevrilmesi ve yerelleştirilmesi, global projelerde iş birliğini kolaylaştırır.

Gelecek Araştırma Yönetimi: RE ve LLM

Açık Araştırma Soruları

- LLM'lerin alan spesifik terminolojiyi ne kadar iyi öğrenebildiği ve transfer öğrenmenin sınırları nedir?
- Halüsinasyonların otomatik tespiti için güvenilir metrikler ve doğrulama mekanizmaları geliştirilebilir mi?
- Prompt tasarımı standartlaştırılabilir mi, yoksa her proje için özel mi olmalıdır?
- İnsan-AI işbirliğinin optimal dengesi hangi faktörlere bağlıdır (proje karmaşıklığı, risk seviyesi, ekip deneyimi)?
- LLM'lerin semantik anlaması, gerçek dünya kısıtlarını (bütçe, zaman, teknoloji limitleri) ne kadar kavrayabilir?

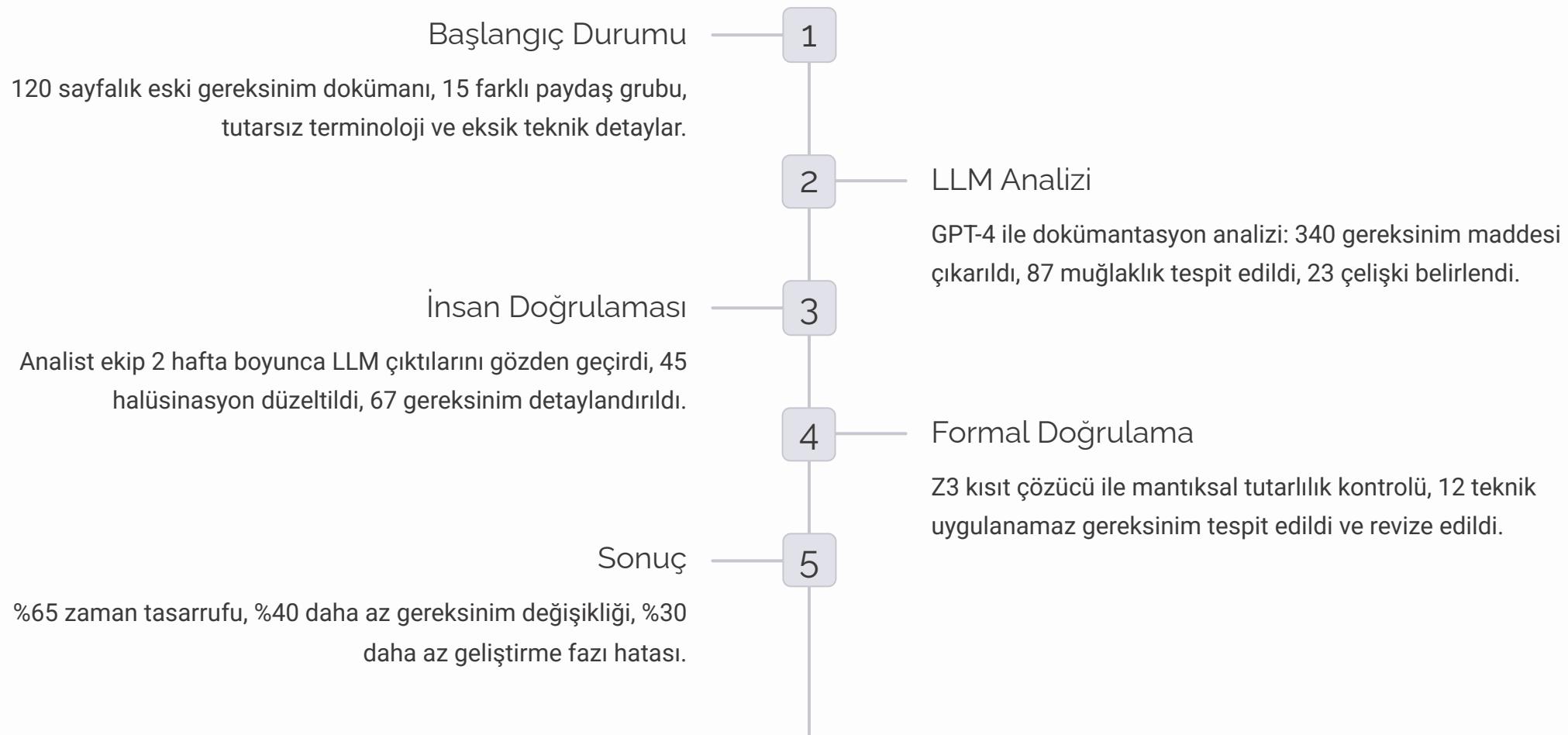
Metodolojik İhtiyaçlar

- RE alanına özel, etiketlenmiş büyük veri setlerinin oluşturulması
- LLM çıktılarının kalitesini ölçen standart metrikler
- İnsan değerlendirmesi protokoller ve inter-rater reliability analizleri
- Hibrit sistemlerin etkinliğini karşılaştıran kontrollü deneyler
- Uzun vadeli etki araştırmaları: LLM kullanımının gereksinim kalitesine, proje başarısına ve ekip dinamiklerine etkisi



Gereksinim Mühendisliği Vaka Çalışması

Bir e-ticaret platformu modernizasyon projesinde LLM destekli RE sürecinin uygulanması:



Bölüm 2 Özeti: Gereksinim Mühendisliğinde Dönüşüm

1 Semantik Devrim

LLM'ler, gereksinimleri sadece metinsel olarak değil, derin semantik ilişkiler içinde işleyebilmekte ve yüksek boyutlu vektör uzaylarında temsil etmektedir.

2 Prompt Mühendisliği

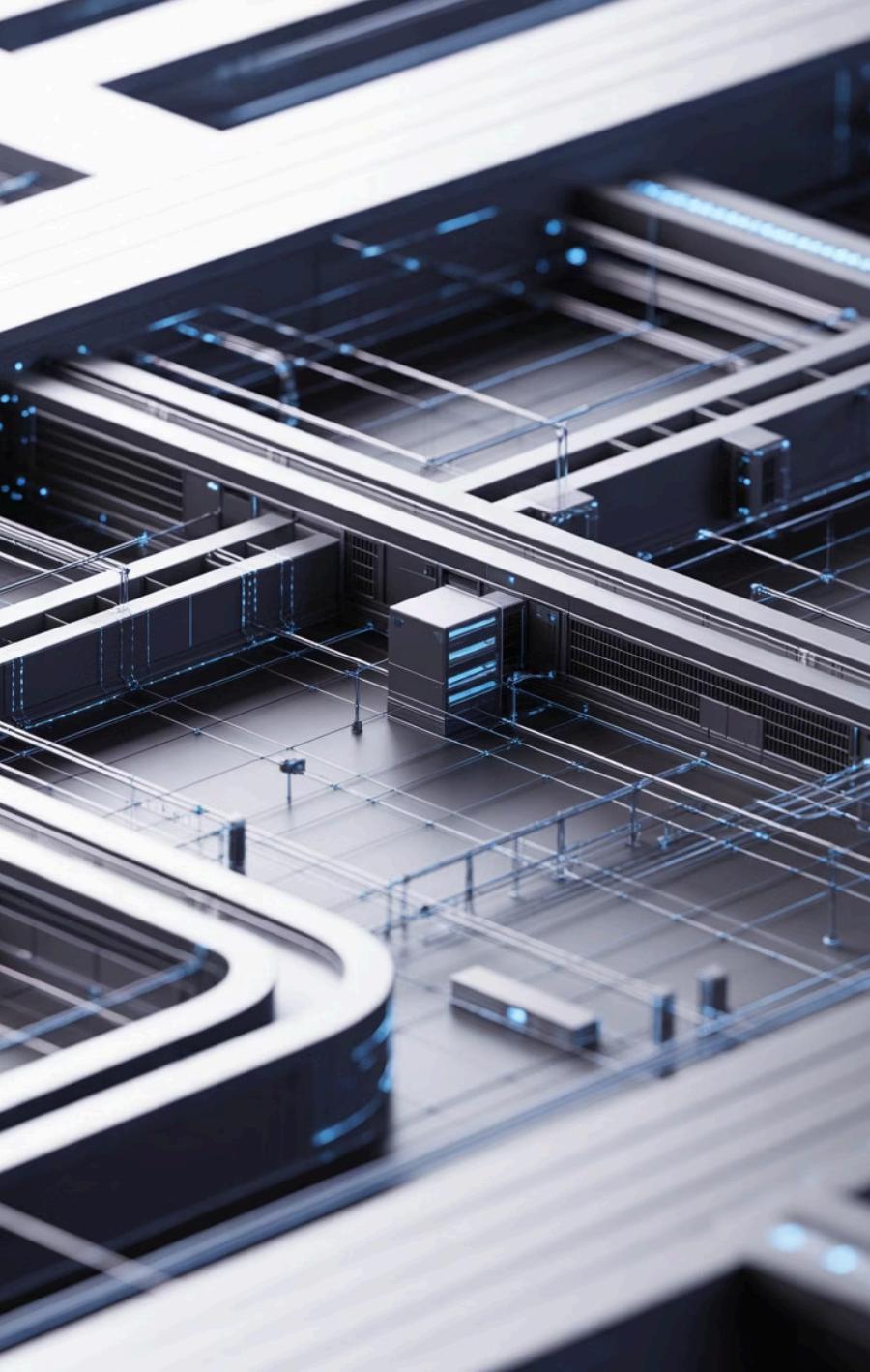
Gereksinim analistlerinin yeni yetkinlik alanı olarak, etkili prompt tasarıımı, few-shot öğrenme ve iteratif iyileştirme stratejileri öne çıkmaktadır.

3 Halüsinasyon ve Doğrulama

Üretken modellerin en büyük riski olan halüsinasyon, human-in-the-loop yaklaşımları ve formal doğrulama araçlarıyla kontrol altına alınmalıdır.

4 Hibrit Gelecek

AI'nın hız ve kapsam sağladığı, insanın ise eleştirel düşünme ve epistemik otorite sağladığı hibrit sistemler, RE'nin geleceğini şekillendirecektir.



Yazılım Mimarisi ve Tasarımın Otomasyonu

Neural Architecture Search ve Yazılım Tasarımı

Yapay zeka alanında, bir problemin çözümü için en uygun sinir ağı yapısını otomatik olarak bulan "Neural Architecture Search" (NAS) teknikleri, yazılım mimarisine de ilham vermektedir. NAS, üç temel bileşenden oluşur:

Arama Uzayı (Search Space)

Olası mimari konfigürasyonlarının tanımlandığı çok boyutlu uzaydır. Yazılım mimarisinde bu, olası modül bölümlemeleri, bağlantı topolojileri ve teknoloji yiğini seçimlerini içerir.

Arama Stratejisi

Evrimsel algoritmalar, pekiştirmeli öğrenme veya Bayesian optimizasyon gibi teknikler kullanılarak optimal mimari aranır.

Performans Değerlendirme

Adayların kalitesi, servisler arası coupling, cohesion, yanıt süresi, kaynak kullanımı gibi metriklerle ölçülür.

Mikroservis Ayrıştırma Problemi: Optimizasyon Yaklaşımı

Monolitik bir uygulamanın mikroservislere bölünmesi, çok kriterli bir optimizasyon problemidir. Amaç, servisler arası bağımlılığı minimize ederken, servis içi tutarlılığı maksimize etmektir:

$$\text{Optimize: } F(S) = \alpha \cdot \text{Cohesion}(S) - \beta \cdot \text{Coupling}(S) - \gamma \cdot \text{Complexity}(S)$$

Burada S, servislerin bir partisyonudur, α , β , γ ise ağırlık katsayılarıdır.

1 Kohezyon (Cohesion)

Bir servis içindeki sınıfların veya fonksiyonların ne kadar ilişkili olduğunu ölçer. Yüksek kohezyon istenir. Metrikler: LCOM (Lack of Cohesion in Methods), semantik benzerlik.

2 Kuplaj (Coupling)

Servisler arasındaki bağımlılık derecesidir. Düşük kuplaj hedeflenir. Metrikler: Servisler arası çağrı sayısı, paylaşılan veri yapıları.

3 Karmaşıklık (Complexity)

Sistemin genel yönetilebilirlik ve anlaşılabilirlik seviyesidir. Çok fazla küçük servis, yönetim karmaşıklığını artırır.

AI algoritmaları (genetik algoritmalar, simülasyonlu tavlama), binlerce olası mimari konfigürasyonunu tarayarak, insan mimarların gözden kaçırabileceği optimal yapıları önerebilir.

Mimari Yeniden Yapılandırma: Legacy Sistem Analizi

Mevcut (legacy) sistemlerin modernizasyonu, yazılım mühendisliğinin en maliyetli alanlarından biridir. Kod tabanının zamanla karmaşıklaması ve dokümantasyonun güncellliğini yitirmesi, sistemin mevcut mimarisinin anlaşılması zorlaştırır.

AI tabanlı "Mimari Yeniden Yapılandırma" araçları, kaynak kodu ve çalışma zamanı verilerini analizerek, sistemin gerçek yapısını ortaya çıkarır. Bu süreç iki aşamalıdır:

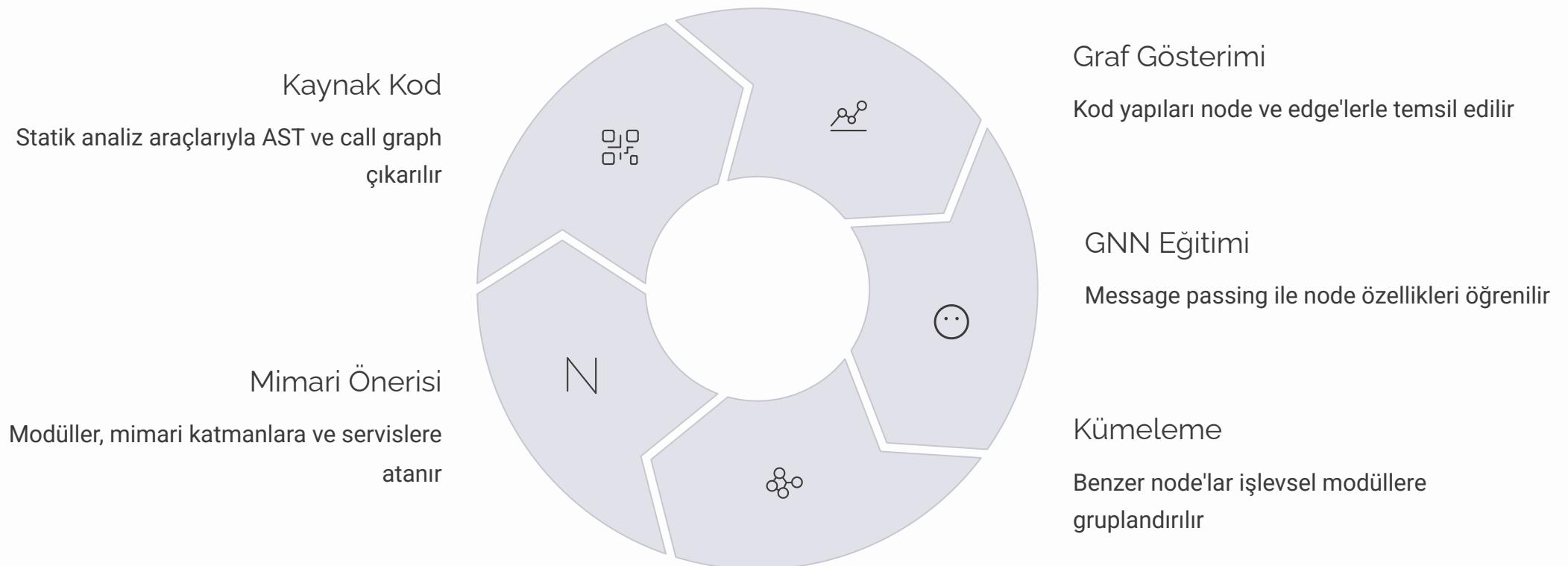
- Architecture Recovery:** Mevcut koddan mimari yapının tersine mühendislikle çıkarılması
- Architecture Reconstruction:** Çıkarılan yapının, modern mimari desenlere uygun şekilde yeniden organize edilmesi



Kümeleme algoritmaları (K-means, DBSCAN, Hierarchical Clustering), sadece statik kod analizine (kim kimi çağrıyor) değil, semantik analize (kod ne yapıyor) de dayanır. Graph Neural Networks (GNN) ve GraphCodeBERT gibi modeller, kodun veri akış grafiğini ve kontrol akış grafiğini analiz ederek, işlevsel olarak ilişkili modülleri grupperler.

Graph Neural Networks ile Mimari Analiz

GNN'ler, yazılım sistemlerini grafikler olarak modelleyerek, node'lar (sınıflar, fonksiyonlar) ve edge'ler (çağrı ilişkileri, veri akışı) arasındaki karmaşık ilişkileri öğrenir.



"ASI-Arch Framework" gibi yapılar, bu analizleri otomatik refactoring önerileriyle birleştirerek, mimari borcu (architectural debt) azaltmayı hedefler.

Üretken Tasarım ve Karar Destek Sistemleri

Üretken yapay zeka (GenAI), sadece var olanı analiz etmekle kalmaz, aynı zamanda yeni mimari tasarımlar da önerir. LLM'ler, geniş tasarım deseni (design patterns) ve mimari stil bilgisiyle eğitilmiştir.



Katmanlı Mimari (Layered)

Sunum, iş mantığı, veri erişimi katmanlarının net ayrimı. Geleneksel kurumsal uygulamalar için uygun.

Avantaj: Basit, anlaşılır, ekip üyelerinin alışık olduğu bir yapı.

Dezavantaj: Yüksek performans gereksinimleri için ölçeklenme zorluğu.



Hexagonal (Ports & Adapters)

İş mantığının dış bağımlılıklardan izole edilmesi. Test edilebilirlik öncelikli projeler için ideal.

Avantaj: Yüksek test edilebilirlik, bağımlılıkların kolayca değiştirilmesi.

Dezavantaj: Fazla soyutlama, küçük projeler için aşırı karmaşık olabilir.



Event-Driven

Olaylara (events) dayalı gevşek bağlı bileşenler. Gerçek zamanlı ve reaktif sistemler için.

Avantaj: Yüksek ölçeklenebilirlik, asenkron işlem, gevşek bağlılık.

Dezavantaj: Hata ayıklama zorluğu, eventual consistency karmaşıklığı.

LLM, bir sistem gereksinimi verildiğinde, bu mimari adayları, her birinin trade-off'larını içeren gerekçeli bir raporla sunar.

Mimari Karar Kayıtları (ADR) ve AI

Mimari Karar Kayıtları (Architecture Decision Records - ADR), bir projede alınan önemli mimari kararları ve bu kararların arkasındaki gerekçeleri belgeleyen yapılandırılmış dokümanlardır. AI, hem ADR oluşturmada hem de mevcut ADR'lerden öğrenmede kullanılabilir.

AI Destekli ADR Üretilimi

LLM'ler, proje bağlamını, kısıtları ve hedefleri anlayarak otomatik ADR taslakları oluşturabilir. Bu taslaklar, mimarların karar verme sürecini hızlandırır ve alternatif değerlendirmesini kolaylaştırır.

Ancak, mimari tasarımın "yaraticılık" ve "stratejik vizyon" gerektiren yönleri, AI için hala zorlu bir alandır. AI, geçmiş verilerdeki desenleri tekrar etme eğilimindedir, bu da radikal yeniliklerin önünü tıkayabilir.

ADR Şablonu Örneği

- Başlık:** Kısa, açıklayıcı karar adı
- Durum:** Önerilen/Kabul Edildi/Reddedildi
- Bağlam:** Hangi problem çözülüyor?
- Karar:** Ne yapılacak?
- Sonuçlar:** Pozitif/negatif etkiler
- Alternatifler:** Değerlendirilen diğer seçenekler

Açıklanabilirlik (Explainability) Sorunu

AI tarafından önerilen karmaşık mimarilerin **açıklanabilirliği** (explainability), kritik bir sorundur. Mimari kararların arkasındaki rasyonelin şeffaf olması, sistemin uzun vadeli bakımı için hayatı önem taşır.



LLM'lerin mimari önerileri genellikle bu piramidin en altında yer alır. Bu nedenle, post-hoc açıklama teknikleri (LIME, SHAP) ve nöro-simbolik yaklaşımlar araştırılmaktadır. Ancak, tam şeffaflık ve yüksek performans arasında trade-off vardır.

Mimari Borç (Architectural Debt) Yönetimi

Teknik borç kavramının mimari boyutu olan "Mimari Borç", sistemin uzun vadeli sağlığını tehdit eden suboptimal tasarım kararlarını ifade eder. AI, mimari borcun tespiti ve yönetiminde etkili bir araçtır.

AI ile Borç Tespiti

- Kod Kokusu Analizi:** God class, spaghetti code gibi anti-pattern'lerin otomatik tespiti
- Kohezyon/Kuplaj Metrikleri:** Zamanla bozulan modül sınırlarının belirlenmesi
- Değişim Analizi:** Sık değişen ve hata üreten modüllerin işaretlenmesi
- Performans Profili:** Darboğaz noktalarının ve ölçeklenme sorunlarının tanımlanması

Önceliklendirme ve Planlama

AI, tespit edilen mimari borç kalemlerini risk, maliyet ve etki analizine göre önceliklendirebilir. Makine öğrenimi modelleri, geçmiş proje verilerinden öğrenerek, hangi borç tiplerinin gelecekte en büyük soruna yol açacağını tahmin edebilir.

Bu sayede, refactoring çalışmaları stratejik olarak planlanır ve kaynaklar en kritik alanlara yönlendirilir.

Mimari Otomasyonunun Sınırları

1 Teknik Sınırlar

AI, sadece eğitim verisindeki kalıpları öğrenebilir. Yeni, benzersiz problemler için tamamen orijinal mimariler üretmekte zorlanır. Ayrıca, performans ve açıklanabilirlik arasındaki trade-off, pratik uygulamaları kısıtlar.

2 Bağlamsal Sınırlar

Mimari kararlar, sadece teknik değil, aynı zamanda iş stratejisi, organizasyonel yapı, ekip becerileri ve pazar dinamikleri gibi faktörlerden etkilenir. AI, bu geniş bağlamı tam olarak kavrayamaz.

3 Epistemik Sınırlar

İyi bir mimari, gelecekteki gereksinimleri öngörme ve esneklik sağlama yeteneği gerektirir. AI'nın olasılıksal doğası, kesin öngörüler yerine olası senaryolar sunar, bu da stratejik kararlar için yeterli olmayabilir.

Sonuç olarak, AI mimari tasarımda güçlü bir karar destek sistemidir, ancak insan mimarin yerini alamaz. Hibrit yaklaşım, optimal sonuçlar verir: AI, alternatif üretir ve analiz eder; insan, bağlamsal bilgi ve stratejik vizyonla nihai kararı verir.

Bölüm 3 Özeti: Mimari Otomasyonunun Dönüşütürücü Gücü

10X

Hızlanma

Mimari adaylarının değerlendirilmesinde zaman tasarrufu

40%

Borç Azalması

Mimari borç tespiti ve önceliklendirme ile

1000+

Konfigürasyonlar

AI'nın tarayabileceği olası mimari konfigürasyonları

Yazılım mimarisi ve tasarımın otomasyonu, NAS tekniklerinden graph neural networks'e, üretken tasarımından mimari borç yönetimine kadar geniş bir yelpazeyi kapsar. AI, mimarların karar verme sürecini hızlandırır, görünmeyen alternatifleri ortaya çıkarır ve uzun vadeli sistemin sağlığını korur. Ancak, bağılamsal anlayış, stratejik vizyon ve yaratıcılık hala insan mimarın domain'i olmaya devam etmektedir. Geleceğin mimari tasarım süreçleri, insan-AI simbiyozunun en başarılı olduğu hibrit modeller olacaktır.

Kod Üretimi ve Sinirsel Program Sentezi

Transformer Mimarisi ve Kodun İstatistiksel Doğası

Kod üretimi (Code Generation), AI4SE alanının en görünür ve etkileyici başarısıdır. GitHub Copilot, Amazon CodeWhisperer, Google Codey gibi araçlar, yazılım geliştirme pratiğini köklü bir şekilde değiştirmektedir. Bu başarının arkasında, Transformer mimarisinin "Attention" (Dikkat) mekanizması yatar.

Doğal dilden farklı olarak, programlama dilleri katı bir sözdizimine (syntax) ve çok uzun vadeli bağımlılıklara (long-range dependencies) sahiptir. Bir değişkenin tanımı ile kullanımı arasında binlerce satır kod olabilir. Bir fonksiyonun çağrııldığı yer, tanımlandığı yerden çok uzak olabilir.

Transformer modelleri (GPT-4, Codex, CodeLlama), Self-Attention mekanizması sayesinde bu uzak ilişkileri modellemede önceki RNN/LSTM mimarilerine göre teorik olarak çok daha üstündür.

Self-Attention Mekanizması

Self-Attention, her token'ın diğer tüm token'larla olan ilişkisini hesaplar:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Bu mekanizma, modelin kodun herhangi bir noktasında, gerekli bağlamı tüm kod tabanından çekmesini sağlar. Örneğin, bir fonksiyon çağrıları yazarken, modelin o fonksiyonun imzasını ve dokümantasyonunu "hatırlaması" mümkün olur.

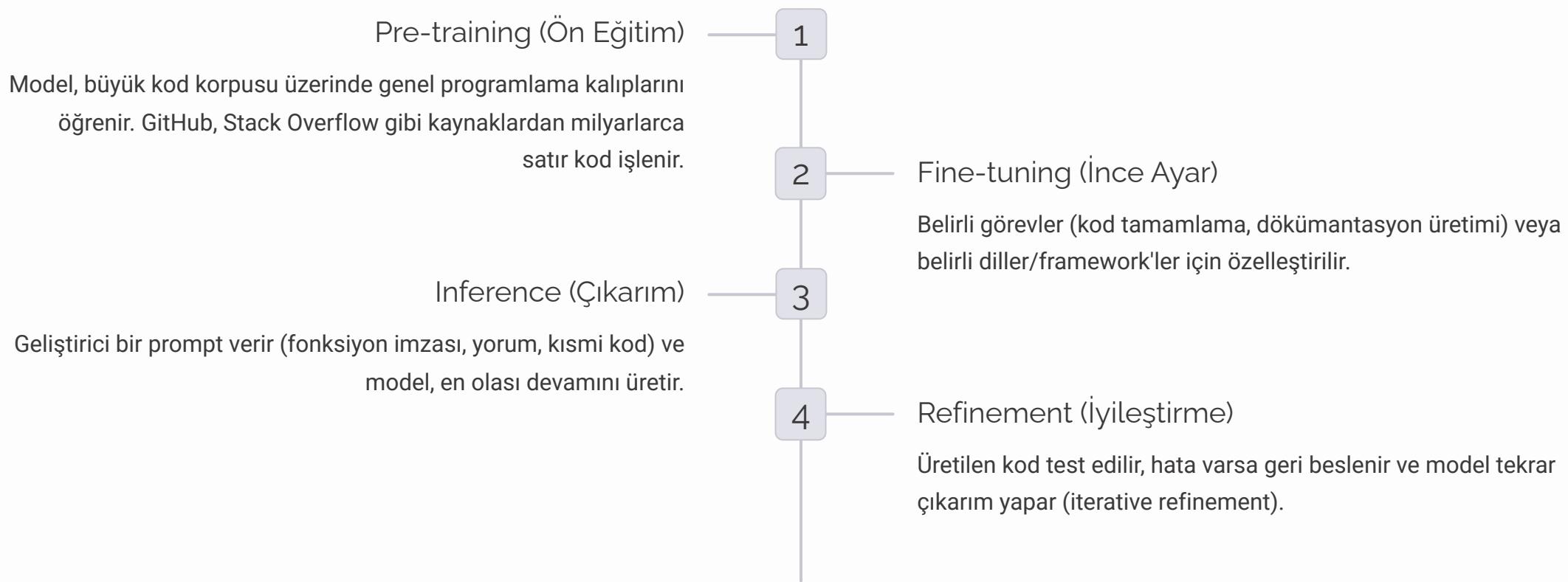
Neural Program Synthesis: Olasılıksal Kod Üretimi

Geleneksel program sentezi, mantıksal spesifikasyonlardan (örneğin giriş-çıkış örnekleri veya formal mantık ifadeleri) program türetme sürecidir. Bu, tümdeğelimli (deductive) veya tümevarımsal (inductive) sentezi teknikleriyle yapılır ve kesin sonuçlar hedefler.

Neural Program Synthesis ise, kod yazmayı bir **koşullu olasılık problemi** olarak ele alır:

$$P(\text{code}|\text{context})$$

Model, milyarlarca satır açık kaynak kod üzerinde eğitilerek, kodun sadece sözdizimini değil, programcıların **niyetini, değişken isimlendirme konvansyonlarını ve yaygın algoritmik kalıpları** öğrenir.

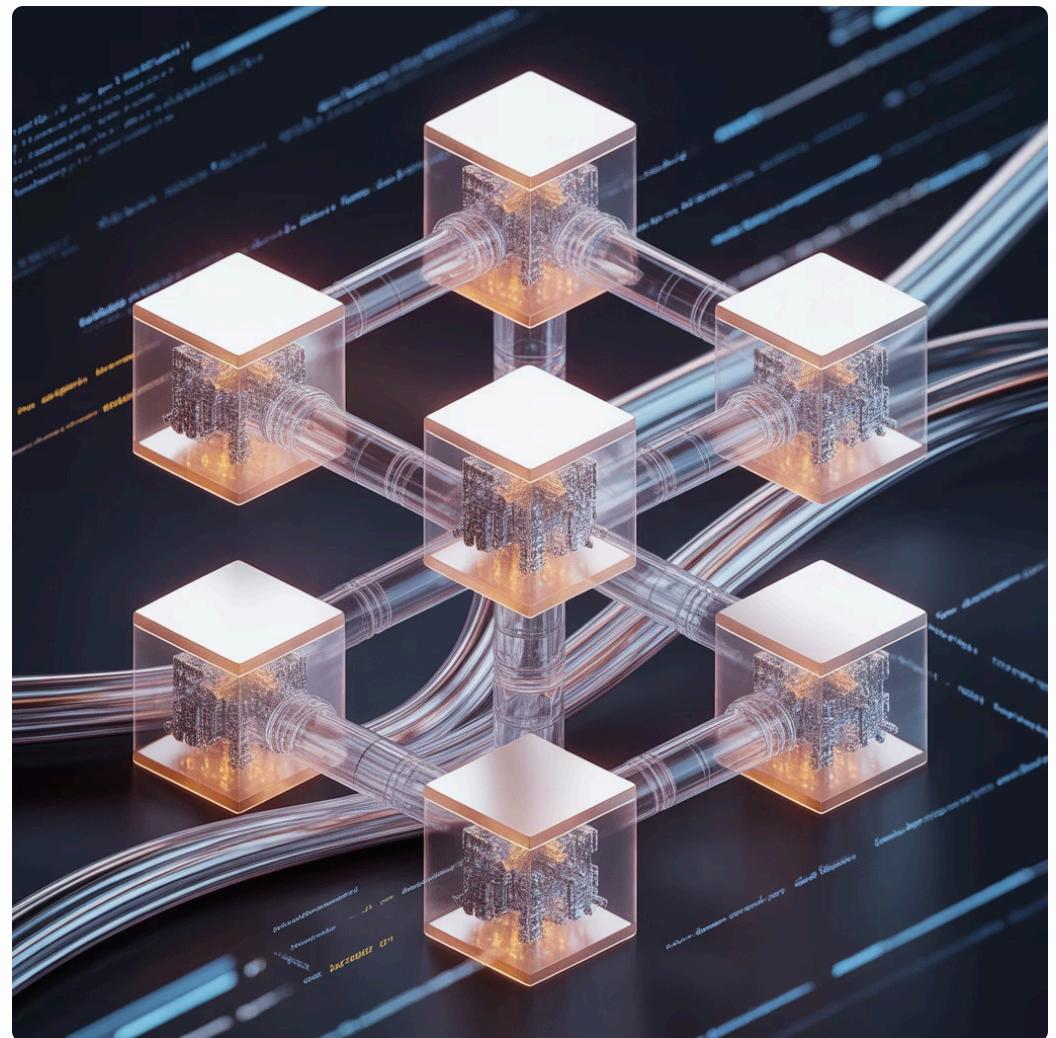


Yapısal Kod Temsili: CodeBERT

Kodun sadece düz metin (token dizisi) olarak işlenmesi, yapısının (AST - Abstract Syntax Tree) ve semantiğinin göz ardı edilmesine neden olabilir. CodeBERT, bu teorik eksikliği gidermek için geliştirilmiş bimodal (doğal dil + programlama dili) bir modeldir.

CodeBERT Ön Eğitim Görevleri

- **Masked Language Modeling (MLM):** Kod ve doğal dildeki token'ların %15'i maskelenir, model bunları tahmin eder
- **Replaced Token Detection (RTD):** Bazı token'lar benzer ancak yanlış token'larla değiştirilir, model hangilerinin değiştirildiğini bulur



CodeBERT, kod-doğal dil eşleştirme görevlerinde (kod arama, dokümantasyon üretimi) state-of-the-art performans göstermiştir. Ancak, kodun veri akışını açıkça modellememesi bir kısıt olarak kalmıştır.

GraphCodeBERT: Veri Akışı Bilinci

GraphCodeBERT, CodeBERT'in üzerine, kodun **veri akış grafiğini** (Data Flow Graph - DFG) de entegre eder. DFG, değişkenlerin nerede tanımlandığını ve değerlerin nasıl aktığıını gösteren yönlendirilmiş bir graftır.



AST (Abstract Syntax Tree)

Kodun sözdizimsel yapısını temsil eder.
Her node bir sözdizimi elemanıdır
(fonksiyon, döngü, ifade vb.)



DFG (Data Flow Graph)

Değişkenler arası veri bağımlılıklarını gösterir. "x = y + 1" ifadesinde, x'in y'ye bağımlı olduğunu edge ile temsil eder.



GraphCodeBERT

Hem token dizisini hem AST'yi hem de DFG'yi birleştiren çok modlu bir transformer. Tüm bu bilgiyi eş zamanlı öğrenir.

GraphCodeBERT, özellikle **kod arama** (code search), **klon tespiti** (clone detection) ve **kod tamamlama** görevlerinde, sadece metin tabanlı modellere göre %5-15 daha yüksek performans göstermiştir. Yapısal bilgi, modelin kodu sadece metin olarak değil, mantıksal bir akış olarak "anlamasını" sağlar.

Kod Üretiminde Pratik Zorluklar



Sözdizimi Hataları

LLM'ler, özellikle uzun kod parçaları üretirken, parantez eşleştirme, noktalı virgül eksikliği gibi sözdizimi hataları yapabilir. Statik analiz araçları ile otomatik düzeltme gereklidir.



Mantıksal Hatalar

Kod sözdizimsel olarak doğru olsa da, mantıksal hata içerebilir (off-by-one, null pointer). Bu hatalar, sadece test ile tespit edilebilir.



Güvenlik Açıkları

Model, eğitim verisindeki güvensiz kod kalıplarını tekrarlayabilir (SQL injection, XSS). Güvenlik tarayıcıları (SAST) entegrasyonu kritiktir.



Lisans ve Telif Hakkı

Eğitim verisinde yer alan kodun lisansları çeşitlidir. Üretilen kodun telif hakkı durumu belirsizdir ve yasal riskler taşır.



Performans ve Verimlilik

Model, işlevsel doğru ancak verimsiz algoritmalar üretебilir ($O(n^2)$ yerine $O(n \log n)$ algoritma kullanılabilir). Performans profileme gereklidir.

Nöro-Sembolik Yaklaşımlar: En İyi İki Dünyanın Birleşimi

Büyük Dil Modelleri (LLM), kod üretiminde etkileyici olsa da, mantıksal tutarlılık konusunda zayıf kalabilirler. "Nöro-Sembolik" (Neuro-symbolic) yaklaşımlar, bu sorunu aşmak için geliştirilmektedir.

Sinirsel (Neural) Bileşen

Sinir ağları, büyük veri setlerinden desen tanıma ve üretkenlik sağlar. EsnekİR, genelleme yapar, olasılıksal çıktılar üretir.

Güçlü yönler: Dil anlama, yaratıcılık, belirsizlikle başa çıkma

Zayıf yönler: Mantıksal tutarlılık, kesin çıkarım, açıklanabilirlik

Sembolik (Symbolic) Bileşen

Mantık programlama, kısıt çözümcüler, formal doğrulama araçları kesinlik ve kanıtlanabilirlik sağlar. Kurallar açık, çıkarımlar deterministik.

Güçlü yönler: Mantıksal çıkarım, formal garanti, açıklanabilirlik

Zayıf yönler: Genelleme, esneklik, ölçeklenme

Nöro-sembolik sistemlerde, LLM kod üretir, sembolik doğrulayıcı kontrol eder. Hata bulunursa, geri besleme ile LLM kodu düzeltir. Bu döngü, hem yaratıcılığı hem de doğruluğu optimize eder.

Nöro-Sembolik Sistemlerin Mimarisi

1. Prompt ve Gereksinim

Geliştirici, doğal dilde veya kısmi kodla niyetini ifade eder. Örnek: "İki sayıyı toplayıp büyük olanı döndüren fonksiyon yaz"

2. Sinirsel Kod Üretimi

LLM, olasılıksal modeline dayanarak bir kod taslağı üretir. Bu kod, semantik olarak doğru ancak detaylarda hatalı olabilir.

3. Sembolik Doğrulama

Üretilen kod, formal doğrulayıcılar (SMT solver, type checker, static analyzer) tarafından kontrol edilir. Sözdizimi, tip uyumu, mantıksal tutarlılık test edilir.

4. Hata Geri Bildirimi

Doğrulayıcı hata bulursa, detaylı hata mesajı LLM'e geri beslenir. Örnek: "Fonksiyon int döndürmelidir, ancak None dönüyor"

5. İteratif İyileştirme

LLM, hata mesajını kullanarak kodu düzeltir. Süreç, doğrulama başarılı olana veya iterasyon limiti dolana kadar devam eder.

6. Test ve Entegrasyon

Doğrulanın kod, birim testlerinden geçirilir ve geliştirici onayından sonra kod tabanına entegre edilir.

DeepLogic: Nöro-Sembolik Program Sentezi

"DeepLogic" gibi sistemler, sinirsel kodlayıcıları mantık programı üretimiyle birleştirerek sembolik çıkarım için teorik garanti sunar. Bu yaklaşım, özellikle güvenlik kritik sistemlerde AI kullanımının önünü açmaktadır.

DeepLogic Çalışma Prensibi

- Gereksinim, mantıksal kısıtlar olarak ifade edilir (formal specification)
- Sinirsel ağ, bu kısıtları karşılayan kod uzayını öğrenir
- Üretilen her kod adayı, sembolik çözücü ile doğrulanır
- Geçerli kodlar arasında, performans ve okunabilirlik kriterlerine göre en iyi aday seçilir

Avantajları

- Garanti edilen doğruluk:** Sembolik doğrulama, kodun spesifikasyona uygunluğunu garantiler
- Açıklanabilirlik:** Mantık kuralları, kodun neden o şekilde üretildiğini açıklar
- Güvenlik:** Kritik sistemlerde kullanılabilir seviyede güvenilirlik

Dezavantaj: Hesaplama maliyeti yüksektir, çünkü her adımda sembolik doğrulama yapılır.

Kod Üretiminde İleri Teknikler



Incremental Code Completion

Token-by-token kod tamamlama yerine, bütün bir fonksiyon veya kod bloğu önerisi. Bağlam farkındalığı kritiktir.



Test Generation

Üretilen kod için otomatik birim test senaryoları oluşturma. Kod kapsamını (coverage) maksimize eden test setleri.



Code Transformation

Mevcut kodu farklı bir paradigmaya (örneğin imperative'den functional'a) veya dile (Python'dan Java'ya) dönüştürme.



Documentation Synthesis

Kod analiz edilerek otomatik dokümantasyon, docstring ve yorum üretimi. Kodun ne yaptığını açıklayan doğal dil metni.

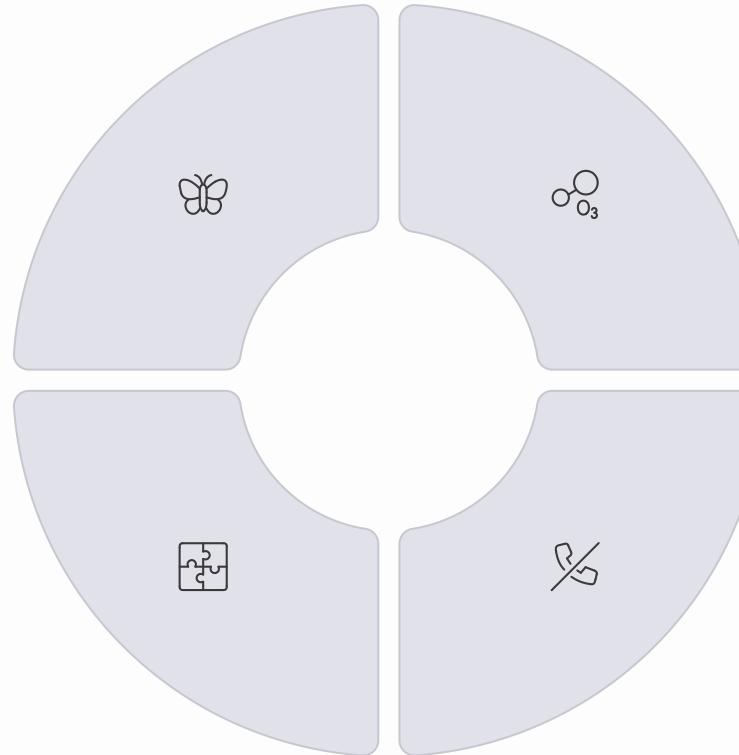
Bölüm 4 Özeti: Kod Üretiminin Teorik ve Pratik Boyutları

Transformer Devrimi

Self-Attention mekanizması, kodun uzun vadeli bağımlılıklarını modelleyerek, RNN/LSTM'in ötesinde performans sağlamıştır.

Pratik Zorluklar

Halüsinasyon, güvenlik açıkları, lisans sorunları ve performans verimliliği, kod üretiminde devam eden araştırma alanlarıdır.



Yapısal Temsil

CodeBERT ve GraphCodeBERT, kodun AST ve DFG gibi yapısal özelliklerini öğrenerek, sadece metin tabanlı modellerden üstün sonuçlar vermiştir.

Nöro-Sembolik Sentez

Sınırsız üretkenlik ile sembolik doğruluğun birleşimi, güvenlik kritik sistemlerde AI kullanımını mümkün kılar.

Kod üretimi, Software 2.0 paradigmının en somut uygulamasıdır. Gelecekte, geliştiriciler kod yazmaktan ziyade, AI'nın ürettiği kodu anlama, doğrulama ve entegre etme rolüne doğru evrilecektir.

Otomatik Program Onarımı (APR) ve Dönüşüm Nöral Makine Çevirisi (NMT) Metaforu

Geleneksel Otomatik Program Onarımı (APR), genellikle arama tabanlı veya kısıt tabanlı yöntemlere dayanıyordu. GenProg gibi araçlar, genetik algoritmalar kullanarak kod üzerinde mutasyonlar yapar ve test senaryoları ile doğrulardı. Ancak bu yöntemler, önceden tanımlanmış şablon ve mutasyon operatörlerine bağımlıydı.

"Neural Program Repair" (NPR) paradigması, program onarımını bir "çeviri" problemi olarak yeniden tanımlar. NMT (Neural Machine Translation) yaklaşımında:

- Kaynak Dil:** Hatalı kod
- Hedef Dil:** Düzeltilmiş kod
- Eğitim Verisi:** GitHub'daki milyonlarca bug fix commit'i

Sequence-to-Sequence (Seq2Seq) modelleri, bu commit geçmişlerinden öğrenerek, hatalı kodların nasıl düzeltildiğine dair genel kalıpları öğrenir.



NMT APR'nin Avantajı

Bu yöntem, önceden tanımlanmış şablonlara ihtiyaç duymadan, çok çeşitli ve karmaşık hataları düzeltme potansiyeline sahiptir. Model, eğitim verisindeki düzeltme stratejilerini genelleyerek, eğitim setinde görülmemiş hata türlerine de uygulanabilir.

APR'de Sequence-to-Sequence Modeller

Seq2Seq mimarisi, encoder-decoder yapısına sahiptir. Encoder, hatalı kodu yüksek boyutlu bir vektör uzayına kodlar (context vector), decoder ise bu vektörden düzeltilmiş kodu üretir.



Ancak, modelin ürettiği yamanın test senaryolarını geçmesi (plausible patch), o yamanın gerçekten doğru olduğu (correct patch) anlamına gelmez. Bu, APR alanındaki temel teorik zorluklardan biridir.

Round-Trip Translation (Gidiş-Dönüş Çeviri)

LLM'lerin APR alanındaki etkinliğini artırmak için geliştirilen ileri tekniklerden biri "Round-Trip Translation" (RTT) stratejisidir. Bu teknik, hatalı kodun önce başka bir temsile çevrilmesi, sonra tekrar orijinal temsile dönüştürülmesi işlemidir.

RTT Süreci

- İlk Çeviri:** Hatalı Python kodu → İngilizce açıklama (veya başka bir dile, örn. Java)
- Geri Çeviri:** İngilizce açıklama → Düzeltilmiş Python kodu

Bu süreç, dil modellerinin "gürültü giderme" (denoising) özelliğinden faydalananır.



Neden İşe Yarar?

LLM'ler, anlamsal olarak tutarlı metinler üretme eğilimindedir. Hatalı kod, ara dile (İngilizce) çevrilirken, semantik anlamı korunur ancak sözdizimsel hatalar kaybolabilir. Geri çeviri sırasında, model semantik anlamdan yola çıkarak doğru kodu üretir.

Bu teknik, özellikle basit sözdizimi hatalarında etkilidir.

Curriculum Learning (Müfredat Tabanlı Öğrenme)

APR modellerinin eğitim sürecinde, eğitim örneklerinin sunulma sırasının model performansına önemli etkisi vardır. "Curriculum Learning" yaklaşımı, insanların öğrenme sürecinden ilham alarak, önce basit hatalardan başlayıp giderek karmaşık hatalara geçmeyi önerir.

1 Seviye 1: Basit Sözdizimi Hataları

Eksik noktalı virgül, yanlış parantez eşleştirme, yazım hataları gibi tek satırlık değişimler

2 Seviye 2: Tip ve Değişken Hataları

Yanlış tip kullanımı, tanımlanmamış değişken, kapsam hataları

3 Seviye 3: Mantıksal Hatalar

Off-by-one, yanlış koşul, eksik kontrol yapıları

4 Seviye 4: Karmaşık Yapısal Hatalar

Çok satırlı değişimler, algoritma tasarım hataları, mimari sorunlar

Bu yaklaşım, modelin öğrenme verimliliğini artırır, eğitim süresini kısaltır ve yerel minimumlara (local minima) saplanma riskini azaltır. Empirik çalışmalar, curriculum learning ile eğitilen APR modellerinin, rastgele sırayla eğitilen modellere göre %10-20 daha yüksek düzeltme başarısı gösterdiğini ortaya koymuştur.

Aşırı Uyum (Overfitting) ve Doğruluk Sorunu

NPR sistemlerinin en büyük teorik kısıtı "aşırı uyum" (overfitting) problemidir. Modeller, eğitim setindeki spesifik düzeltme kalıplarını ezberleyebilir, ancak eğitim verisinde görülmemiş yeni tip hataları düzeltmekte başarısız olabilir.

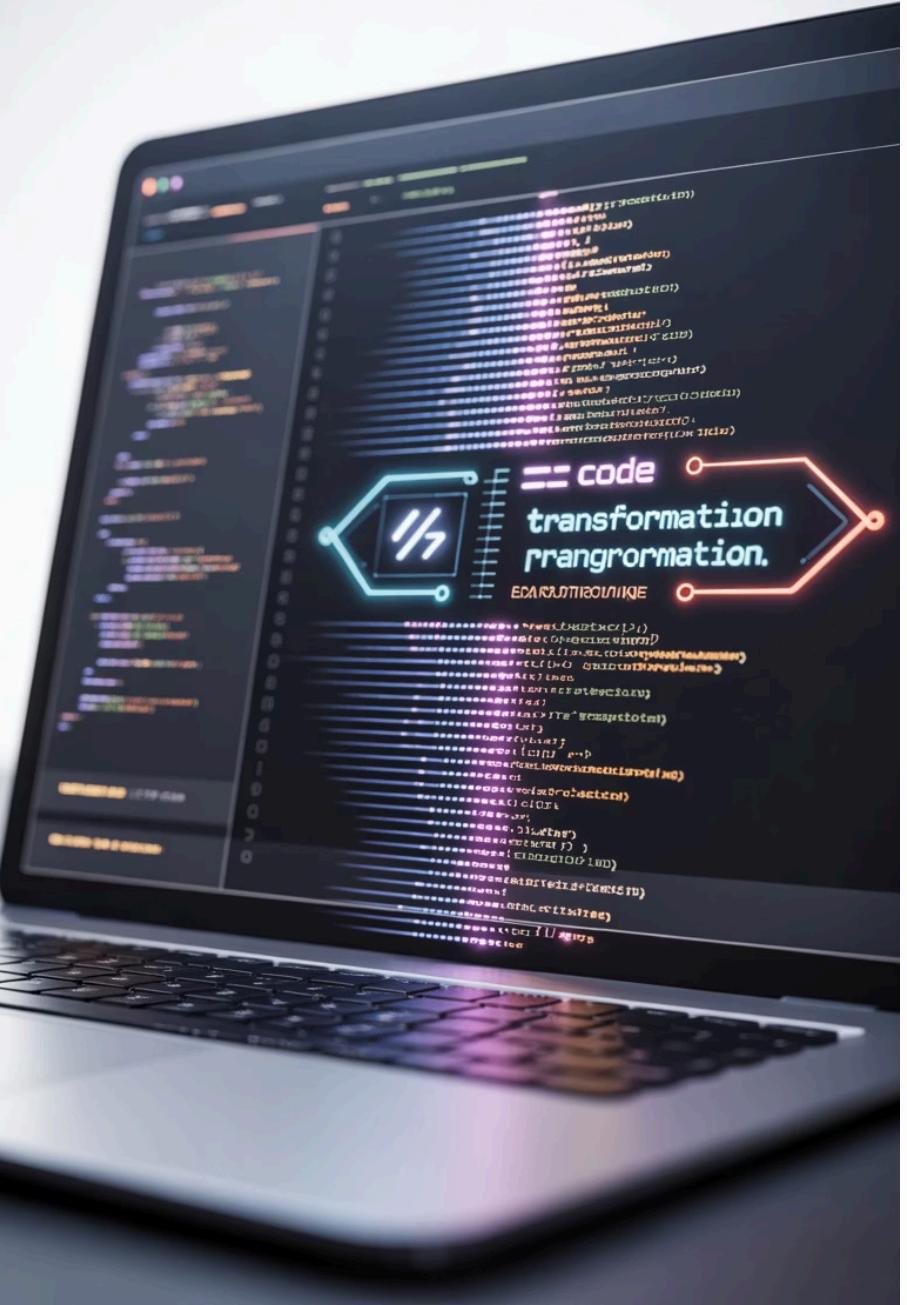
1 Plausible Patch (Makul Yama)

Modelin ürettiği yama, mevcut test senaryolarını geçer. Ancak bu, yamanın tamamen doğru olduğunu garanti etmez. Test senaryoları eksik veya yüzeysel olabilir.

2 Correct Patch (Doğru Yama)

Yama, sadece testleri değil, aynı zamanda geliştiricinin gerçek niyetini de karşılar. Tüm köşe durumlarını ve gelecekteki kullanım senaryolarını kapsar.

Araştırmalar, NPR araçlarının ürettiği yamaların %30-60'ının plausible ancak yanlış olduğunu göstermiştir. Model, testleri kandıran ancak sistemin başka yerlerini bozan veya yeni hatalar yaratan yamalar üretebilir. Bu durum, APR alanında hala çözülmesi gereken temel bir güvenilirlik sorunudur ve insan doğrulamasının gerekliliğini vurgular.



Kod Dönüşümü: Transpilation ve Modernizasyon

Program onarımının yanı sıra, AI kod dönüşümü (code transformation) alanında da büyük ilerleme kaydetmiştir. Bu, bir programlama dilinden diğerine çeviri (transpilation) veya eski kodun modern standartlara uyarlanması (modernization) işlemlerini kapsar.

Transpilation Örnekleri

- Python 2 → Python 3
- JavaScript ES5 → ES6+
- Java → Kotlin
- C → Rust
- Legacy COBOL → Modern Java

Bu dönüşümler, sadece sözdizimsel çeviri değil, aynı zamanda deyimsel (idiomatic) kod yazımını da içerir. Örneğin, Python'daki list comprehension yapısını Java Stream API'sine dönüştürmek.

Modernizasyon Boyutları

- **Sözdizimsel:** Eski dil özelliklerinin yeni sürüm eşdeğerlerine dönüştürülmesi
- **Semantik:** Fonksiyonel eşdeğerliliğin korunması
- **Performans:** Modern dillerin optimizasyon avantajlarından faydalama
- **Güvenlik:** Bilinen güvenlik açıklarının giderilmesi

APR ve Kod Dönüşümünde Gelecek Yönlemleri



Pekiştirmeli Öğrenme (RL)

APR'yi bir RL problemi olarak modelleme: Her düzeltme adımı bir eylem, test sonuçları ödül/ceza sinyali olarak kullanılır.



İnteraktif APR

Geliştirici ile model arasında gerçek zamanlı diyalog: Model soru sorar, geliştirici netleştirir, model rafine eder.



Çok Modlu (Multimodal) APR

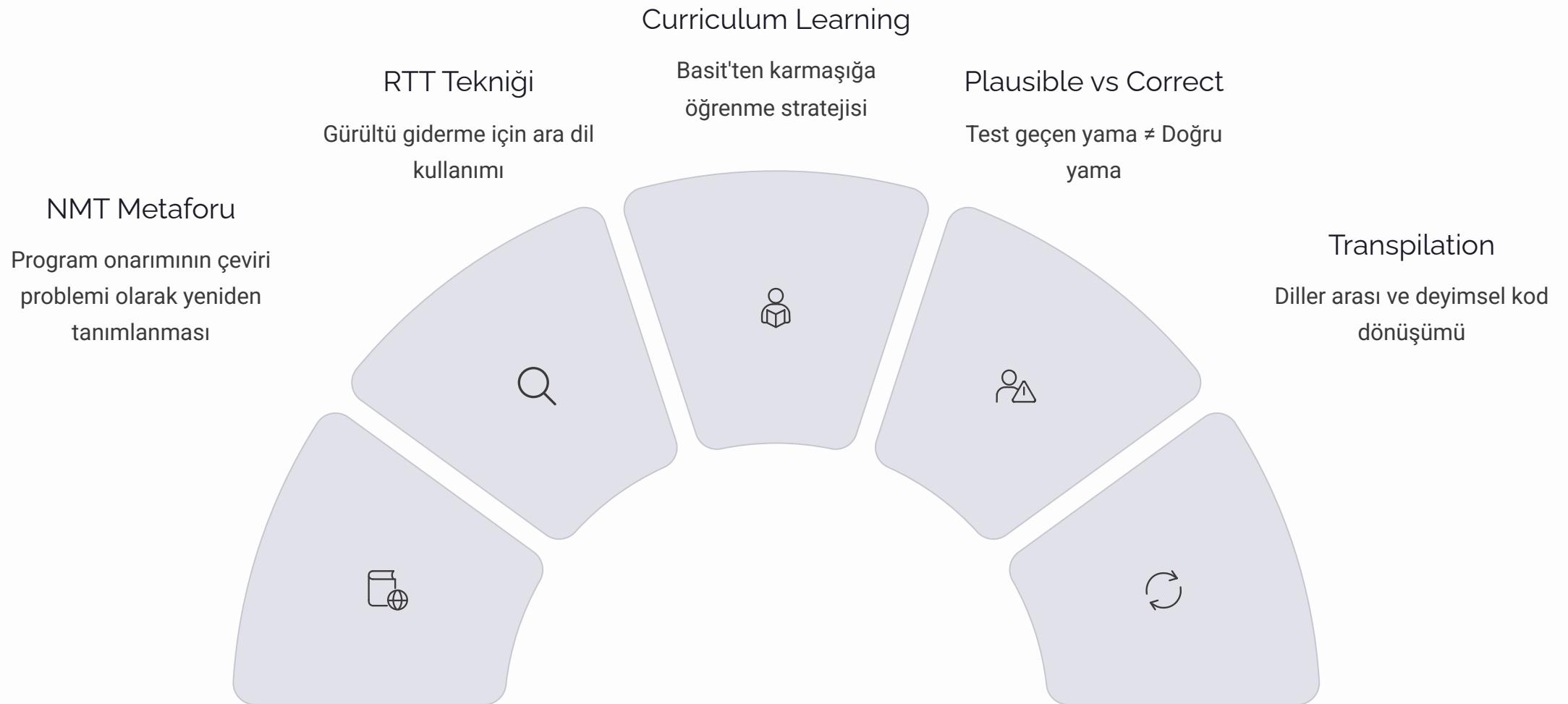
Kod, doküman, commit mesajları, issue tracker kayıtları birlikte işlenerek bağlam zenginleştirilir.



Formal Garanti

Nöro-sembolik yaklaşımlarla, üretilen yamaların belirli özellikleri garanti etmesi (örn. "Bu yama deadlock yaratmaz")

Bölüm 5 Özeti: APR ve Dönüşümün Dönüştürücü Potansiyeli



Otomatik program onarımı ve kod dönüşümü, yazılım bakımının en maliyetli yönlerini otomatikleştirme potansiyeline sahiptir. Ancak, aşırı uyum, plausible-correct ayrımı ve güvenilirlik sorunları, bu alanda devam eden araştırma ihtiyacını göstermektedir. Geleceğin APR araçları, nöro-sembolik yaklaşımalar ve formal doğrulama entegrasyonuyla daha güvenilir olacaktır.

Test, Doğrulama ve Oracle Problemi

Test Oracle Problemi: Epistemolojik Bir Engel

Yapay zeka sistemlerinin ve AI tarafından üretilen kodların test edilmesindeki temel teorik engel "Test Oracle Problemi"dir. Bir test oracle'ı, verilen bir girdi için **beklenen çıktıyı belirleyen mekanizmadır**.

Klasik yazılım testinde, oracle genellikle şunlardan biri olabilir:

- **Spesifikasyon:** Formal gereksinim dokümanı
- **Referans Uygulama:** Doğru çalıştığı bilinen başka bir sistem
- **Manuel Doğrulama:** İnsan uzman kontrolü
- **Invariantlar:** Her zaman doğru olması gereken özellikler

Ancak makine öğrenimi modelleri, simülasyonlar, optimizasyon algoritmaları veya arama motorları gibi sistemler için, her girdi için doğru cevabı önceden bilmek genellikle imkansızdır. Bu tür sistemlere "**non-testable programs**" (test edilemez programlar) denir.

Örnek: Bir arama motorunun "yapay zeka eğitimi" sorusu için döndürdüğü sonuçların "tamamen doğru" olup olmadığına karar vermek subjektif ve bağlam bağımlıdır.

Metamorfik Test: Oracle Eksikliğine Çözüm

Bu sorunu aşmak için geliştirilen "Metamorfik Test" (Metamorphic Testing - MT) teorisi, test paradigmalarında köklü bir değişim yaratmıştır. MT, beklenen çıktıının değerini kontrol etmek yerine, **girdilerdeki değişikliklerin çıktılarında yaratması gereken değişim ilişkilerini** (Metamorphic Relations - MRs) kontrol eder.

$f(x_1)$ ve $f(x_2)$ arasındaki ilişki biliniyor \Rightarrow Oracle gerektirmeden test yapılabilir

Matematiksel Örnek: Sinüs Fonksiyonu

Bir $f(x) = \sin(x)$ fonksiyonunu test ederken, $f(1.234)$ değerini bilmeyebiliriz (oracle yok).

Ancak, trigonometrik özellik:

$\sin(\pi - x) = \sin(x)$ biliniyor (MR).

Test: Eğer program $f(\pi - 1.234) \neq f(1.234)$ veriyorsa, hata tespit edilir.

Makine Öğrenimi Örneği: Sınıflandırıcı

Bir görüntü sınıflandırıcısı test edilirken, doğru etiketi bilinmeyen bir resim kullanılabilir.

MR: Resim hafifçe döndürüldüğünde, sınıf etiketi değişimmemelidir (rotasyon invariansı).

Test: Orijinal resim "köpek" sınıfına atanmışsa, 15° döndürülmüş versiyonu da "köpek" olmalıdır.

Arama Motoru Örneği

Arama sonuçlarının doğruluğunu bilmek zor, ancak bazı özellikler kontrol edilebilir.

MR: Soru kelimeleri farklı sırada girilirse, sonuçların içeriği (sıra değişebilir ancak içerik) aynı olmalıdır.

Test: "makine öğrenimi kitapları" ve "kitapları makine öğrenimi" sorguları benzer sonuçlar vermelidir.



Metamorfik İlişkilerin Tasarımı

Metamorfik test etkinliği, doğru MR'lerin tasarlanmasına bağlıdır. İyi bir MR, hem doğrulanabilir olmalı hem de gerçekten hataları yakalayabilmelidir.

MR Kategorileri

- Değişmezlik (Invariance):** Bazı girdiler değiştiğinde çıktı sabit kalır (örn. permütasyon invariansı)
- Eşdeğerlik (Equivalence):** Farklı girdiler aynı çıktı üretir
- Orantısalılık (Proportionality):** Girdideki değişim, çıktıda belirli bir orana göre değişim yaratır
- Toplamsallık (Additivity):** $f(x + y) = f(x) + f(y)$ gibi ilişkiler
- Monotoniklik:** Girdi artarsa çıktı da artar veya azalır

MR Geçerliliği

Bir MR'nin geçerli olması için:

- Alan Bilgisi:** MR, test edilen sistemin alanına uygun olmalıdır
- Mantıksal Tutarlılık:** MR matematiksel veya mantıksal olarak kanıtlanabilir olmalıdır
- Uygulama Genelligi:** Çok spesifik değil, geniş bir girdi kümlesi için geçerli olmalıdır
- Hata Yakalama Gücü:** Gerçek hataları tespit edebilmelidir, triviallık tuzağına düşmemelidir

Fuzzing ve AI Destekli Test Verisi Üretimi

Yapay zeka, test verisi üretiminde de devrim yaratmaktadır. Geleneksel "Fuzzing" teknikleri rastgele girdiler üretirken, AI destekli fuzzing, kodun daha derinlerine ulaşmak ve çökme (crash) oluşturmak için girdileri akıllıca mutasyona uğratır.

Geleneksel Fuzzing

Tamamen rastgele veya basit mutasyonlarla (bit flip) girdi üretimi. Kod kapsamı (coverage) sınırlıdır.

ML Destekli Fuzzing

Makine öğrenimi modelleri, hangi mutasyonların yeni kod yollarına ulaştığını öğrenir ve stratejik girdiler üretir.



Kapsama Güdümlü Fuzzing

AFL (American Fuzzy Lop) gibi araçlar, kod kapsamını maksimize edecek girdileri seçer. Genetik algoritma prensipleriyle çalışır.

RL Tabanlı Fuzzing

Pekiştirmeli öğrenme ajanları, ödül sinyali olarak kod kapsamı ve çökme tespitini kullanarak optimal fuzzing stratejisi öğrenir.

AI destekli fuzzing, geleneksel yöntemlere göre %30-50 daha fazla kod kapsamına ulaşabilmekte ve %20-40 daha fazla benzersiz bug tespit edebilmektedir.

Karar Verilemezlik (Undecidability) ve Teorik Sınırlar

AI tarafından üretilen kodların doğrulanması, teorik bilgisayar biliminin temel sınırlarına çarpar. **Rice Teoremi**, programların özellikleri hakkında algoritmik karar vermenin sınırlarını tanımlar:

"Bir programın herhangi bir **net olmayan** (non-trivial) semantik özelliğinin algoritmik olarak kesin bir şekilde belirlenmesi imkansızdır."

Net Olmayan Özellik Nedir?

Bir özellik "net olmayan" (non-trivial) ise, bazı programlar bu özelliğini taşıırken bazıları taşımaz. Örnekler:

- "Bu program hatasız mıdır?"
- "Bu program sonlu zamanda durur mu?" (Halting Problem)
- "Bu program sonsuz döngüye girer mi?"
- "Bu program null pointer exception fırlatır mı?"

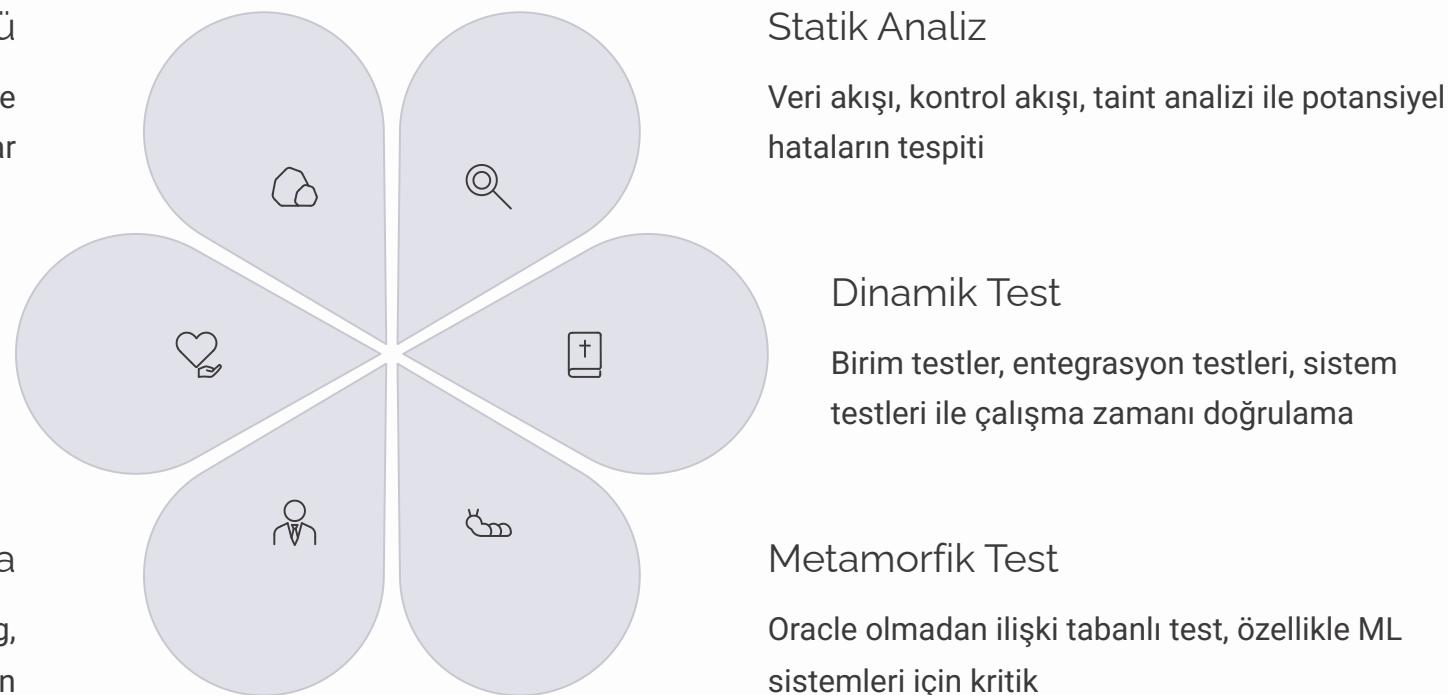
Pratik İmplikasyonlar

Bu matematiksel gerçek, LLM'lerin ürettiği kodun %100 doğru olduğunu garanti eden tam otomatik bir sistemin **teorik olarak imkansız** olduğunu gösterir.

Statik analiz araçları ve formal yöntemler, ancak belirli soyutlamalar ve varsayımlar altında "yaklaşık" sonuçlar verebilir. Örneğin, tip sistemleri bazı hataları yakalayabilir, ancak tüm mantıksal hataları tespit edemez.

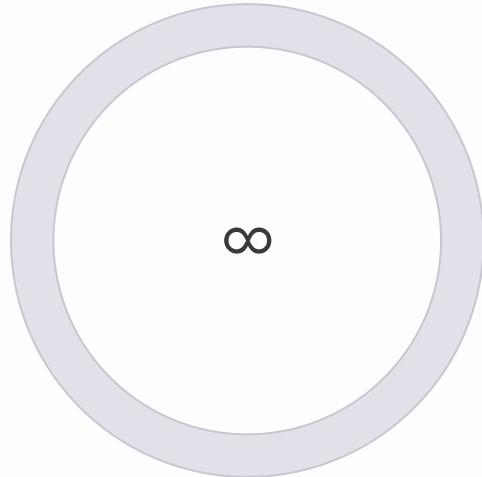
Hibrit Doğrulama Stratejileri

Karar verilemezlik sorununun pratik çözümü, çoklu doğrulama tekniklerinin birleşimidir. Hiçbir teknik tek başına yeterli değildir, ancak birlikte yüksek güven sağlayabilirler.



Bu teknikler, farklı hata türlerini yakalama konusunda tamamlayıcıdır. Örneğin, tip kontrolü sözdizimsel hataları, statik analiz mantıksal akış hatalarını, dinamik test çalışma zamanı davranışını, insan incelemesi ise bağlamsal uygunluğu değerlendirir.

Bölüm 6 Özeti: Test ve Doğrulamanın Epistemolojik Boyutu



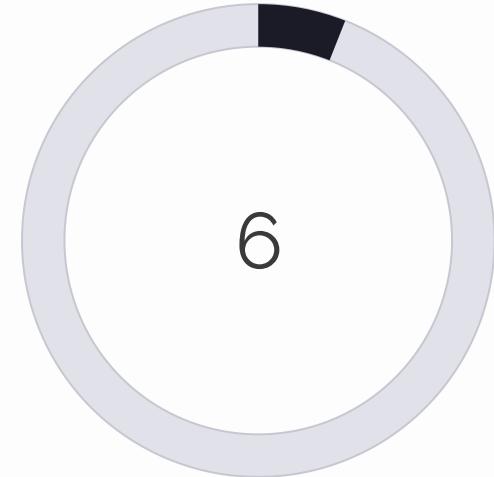
Oracle Problemi

Doğru cevabın bilinmediği sistemler için test zorluğu



Rice Teoremi

Programların net olmayan özelliklerinin kesin olarak belirlenemezliği



Hibrit Doğrulama

Tamamlayıcı doğrulama tekniklerinin birleşimi

Test Oracle Problemi ve Rice Teoremi, AI üretimi kodların tam otomatik doğrulamasının teorik sınırlarını çizmektedir. Metamorfik test, bu sınırları aşmak için güçlü bir araç sunarken, hibrit doğrulama stratejileri pratik güvenilirlik sağlar. "İnsan-Döngüde" yaklaşımı, sadece etik bir tercih değil, matematiksel bir zorunluluktur. Gelecekteki test araçları, AI destekli otomasyon ile insan uzmanlığını birleştiren hibrit sistemler olacaktır.

Sosyo-Teknik Dinamikler ve Epistemolojik Riskler

Bilişsel Yük Teorisi ve AI Pair Programming

Yapay zeka asistanlarının (GitHub Copilot, ChatGPT, vb.) yazılım geliştirmeye entegrasyonu, geliştiricilerin bilişsel süreçlerini derinden etkiler. Bilişsel Yük Teorisi (Cognitive Load Theory - CLT) perspektifinden bakıldığından, bu araçlar iki yönlü bir etki yaratır:

İçsel Yükün (Intrinsic Load) Azalması

AI, sözdizimi hatırlama, boilerplate kod yazma, API dokümantasyonu arama gibi rutin ve düşük seviyeli görevleri üstlenerek, geliştiricinin zihinsel kapasitesini daha karmaşık problemlere (mimari tasarım, algoritma seçimi, edge case analizi) ayırmasını sağlar.

Pozitif etki: Geliştirici, "nasıl yazacağım" yerine "ne yapmalıyım" sorusuna odaklanabilir.

Dışsal Yükün (Extraneous Load) Artması

Geliştirici, AI tarafından üretilen kodu okumak, anlamak ve doğrulamak zorundadır. Kendi yazmadığı bir kodu incelemek, genellikle sıfırdan yazmaktan daha fazla bilişsel efor gerektirebilir, çünkü kodun arkasındaki niyeti çıkarsamak gereklidir.

Negatif etki: "Bu kod ne yapıyor? Neden böyle yazdı? Güvenebilir miyim?" soruları sürekli sorulur.

AI Pair Programming: İnsan-İnsan vs İnsan-AI Dinamikleri

Pair programming, iki geliştiricinin bir görevde birlikte çalışmasıdır. AI pair programming ise, bir geliştiricinin AI asistanı ile çalışmasıdır. Bu iki durum arasında önemli farklar vardır:

Boyut	İnsan-İnsan Pair Programming	İnsan-AI Pair Programming
İletişim	Karşılıklı konuşma, soru-cevap, tartışma	Tek yönlü prompt, AI açıklama yapmaz
Paylaşılan Anlayış	Ortak zihinsel model (shared mental model) gelişir	AI'nın modeli opak, geliştirici tahmin yürütür
Sosyal Dinamik	Empati, güven, sosyal normlara uyum	Sosyal boyut yok, araçsal ilişki
Hata Düzeltme	Anlık geri bildirim, müzakere	Geliştirici manuel düzeltir veya prompt'u değiştirir
Bilgi Transferi	Çift yönlü öğrenme	AI'dan geliştiriciye tek yönlü (AI öğrenmez)
Güven Mekanizması	İtibar, geçmiş deneyim, sosyal sinyal	Model performansı, doğrulama, şüphecilik

İnsan partnerle çalışırken sosyal iletişim ve paylaşılan zihinsel modeller ön plandayken, AI partnerle çalışırken "şüphecilik" ve "doğrulama" mekanizmaları daha kritik hale gelir.

Teknoloji Kabul Modeli (TAM) ve AI Araçları

Geliştiricilerin AI araçlarını benimsemesi, Teknoloji Kabul Modeli (Technology Acceptance Model - TAM) çerçevesinde incelenmektedir. TAM'a göre, bir teknolojinin kullanımını belirleyen iki temel faktör vardır:



Algılanan Fayda (Perceived Usefulness)

Aracın iş performansını artırdığına inanma derecesi. Eğer geliştirici AI'nın kendisini daha hızlı ve verimli yaptığına inanırsa, kullanım artar.

AI için: Kod tamamlama hızı, hata oranı azalması, öğrenme eğrisinin kısalması gibi metrikler önemlidir.



Algılanan Kullanım Kolaylığı (Perceived Ease of Use)

Aracın ne kadar kolay öğrenilip kullanılabildiği. Eğer AI aracı karmaşık, anlaşılmaz veya tutarsızsa, benimseme direnci oluşur.

AI için: Prompt yazma yükü, aracın entegrasyon karmaşıklığı, öğrenme maliyeti belirleyicidir.

Ancak AI araçları için iki ek boyut kritiktir:

Güven (Trust)

Geliştiricinin AI'nın ürettiği kodun kalitesine, güvenliğine ve doğruluğuna güvenme derecesi. Güven, geçmiş deneyimlerle (AI daha önce kaç kez hatalı kod üretti?) ve açıklanabilirlikle ilişkilidir.

Açıklanabilirlik (Explainability)

AI'nın neden o kodu önerdiğini anlama yeteneği. Black box modeller, geliştiricilerin güvenini zayıflatır. "Bu kodu neden önerdin?" sorusuna yanıt verebilme, benimseme için kritiktir.

Epistemolojik Kriz: "Vibe Coding" Fenomeni

Yapay zekanın yazılım mühendisliğine girişи, ciddi epistemolojik (bilgi kuramsal) riskleri beraberinde getirir. "Vibe Coding" olarak adlandırılan yeni fenomen, geliştiricilerin kodun detaylarına hakim olmadan, sadece üst seviye niyetlerle ve AI'nın "hissiyatına" güvenerek kodlama yapmasını ifade eder.

Anlama İllüzyonu (Illusion of Understanding)

Geliştirici, AI'nın ürettiği kodu yüzeysel olarak inceleyip "mantıklı görünüyor" diyerek kabul eder, ancak kodun derin mantığını veya köşe durumlarını anlamaz. Bu durum, "**şemsiye altında kalmak**" metaforuyla açıklanabilir: Yağmur yağdığını görüp sizi koruyan şemsiyeniz sizi koruyor, ancak şemsiyenin mekanik yapısını veya su geçirmezlik prensibini anlamıyorsunuz.

Stokastik Papağanlar

LLM'ler, istatistiksel korelasyonlara dayalı çıktılar üretir. Anlamdan yoksun, sadece olası token dizilerini tahmin ederler. Bu nedenle, "Stokastik Papağanlar" olarak adlandırılırlar. AI'nın ürettiği kod, anlamlı görünse de, gerçek bir kavrayışın ürünü değildir.

Sonuç: Geleceğin Yazılım Mühendisliği - Hibrit Zeka Dönemi

Software 2.0: Geri Dönülemez Dönüşüm

Bu sunumda incelenen teorik ve pratik gelişmeler, yazılım mühendisliğinin Software 1.0'dan Software 2.0'a doğru geri dönülemez bir dönüşüm içinde olduğunu göstermektedir. Bu yeni paradigma, deterministik kuralların yerini olasılıksal modellerin, açık kodlamadan yerini veri kürasyonunun aldığı bir dünyayı temsil etmektedir.

10X

Verimlilik Artışı

AI destekli araçlarla geliştirme hızında potansiyel kazanç

60%

Otomasyon Oranı

Rutin kodlama görevlerinin otomasyonu

∞

Epistemik Sorumluluk

Anlama ve doğrulamanın önemi hiç bitmesin

Metodolojik Dönüşüm

Gereksinimden teste kadar tüm SDLC, üretken AI ve veri odaklı tekniklerle yeniden şekillenmektedir. Yazılım artık sadece inşa edilen değil, optimize edilen ve öğrenen bir varlıktır.

Teorik Derinleşme

Vektör uzayları, türevlenebilir programlama, metamorfik test, Rice Teoremi gibi kavramlar, yazılım mühendisliğinin teorik zeminini matematize etmekte ve disiplini bilimsel olarak güçlendirmektedir.

İnsan Faktörü

Geliştiricinin rolü "kodlayıcı"dan "sistem küratörü", "doğrulayıcı" ve "AI yöneticisi"ne evrilmektedir. Prompting, model seçimi, hibrit doğrulama yeni yetkinlikler olarak öne çıkmaktadır.

Epistemik Sorumluluk

Otomasyonun getirdiği kolaylık, anlama yetisinin kaybına yol açmamalıdır. İnsan-Al simbiyozu, eleştirel düşünme ve etik sorumluluk ekseninde kurulmalıdır. Vibe coding'den uzak durulmalıdır.

Geleceğin yazılım mühendisliği, algoritmik yetkinlik kadar, yapay zeka modellerini yönetme, doğrulama ve onlarla etkili iletişim kurma becerilerini de gerektirecektir. Bu dönüşüm, disiplinin bilimsel temellerini zayıflatmak yerine, istatistik ve optimizasyon teorileriyle güçlendirerek daha karmaşık problemlerin çözümüne olanak tanıyacaktır.

"Yazılım mühendisliğinin geleceği, insanın yaratıcılığı ile yapay zekanın hesaplama gücünün simbiyozunda yatmaktadır. Ancak bu simbiyozun sağlığı olması için, epistemik otoritenin insanda kalması ve eleştirel düşüncenin hiçbir zaman terk edilmemesi gerekmektedir."