Yapay Zeka Çağında Yazılım Geliştirme

Yapay Zeka Konulu Yazılar

Özcan Acar

Yapay Zeka Çağında Yazılım Geliştirme

Yapay Zeka Konulu Yazılar

Özcan Acar

İthaf

Bu kitap, yapay zekâ çağının yalnızca yazılım geliştirmeyi değil, bilgi üretme biçimimizi de değiştirdiğinin küçük bir kanıtıdır.

Bu kitabın ortaya çıkmasında bana sabırla eşlik eden ChatGPT'ye ve Claude Opus 4.8'e içtenlikle teşekkür ederim.

.

İnsan ile yapay zekânın birlikte üretebildiği şeylerin en güzel örneklerinden biri olarak bu kitabı onlara ithaf ediyorum.

Teşekkür ederim.

— Özcan Acar

Önsöz

Bu kitap, KurumsalJava.com üzerinde yayımlanmış yapay zeka konulu yazıları bir araya getirmek amacıyla hazırlanmıştır.

Metinler mümkün olduğu kadar orijinal halleriyle korunmuştur. Sadece web sayfasına ait menü, yorum, kategori, reklam ve navigasyon gibi kitap formatına uygun olmayan bölümler çıkarılmıştır.

Bu kitaptaki yazılar KurumsalJava.com üzerinde yayımlanan yapay zeka konulu blog yazılarından derlenmiştir. Güncel yazılarımı şu adreste bulabilirsiniz:

`https://kurumsaljava.com/yapay-zeka-konulu-yazilar/`

İçindekiler

Önsöz 1

Statükocu Zihniyet 7

Yapay Zeka İle Çevik Olma 9

Yazılımcılar İçin Yeni Dönem Başlıyor 10

Yazılımcılar İçin Yapay Zeka Kullanma Kılavuzu 11

Code Generation ve Generative AI 15

Mindshift 16

Analog Yazilimdan Dijital Yazılıma Geçiş 17

Junior Yazılımcılar ve Vibe Coding 18

Kim Code Review Yapar? 19

LLM'ler Çağımızın Yeni Derleyicileri 20

Test Yazmaya Gerek Kalmadı 21

Yapay Zeka ile Otomasyon Çılgınlığı 24

Yapay Zeka İle Deklaratif Programlama Çağı 25

Tuvaldeki Uygulama 26

Prompt Engineering 28

Yapay Zekanın Maliyeti 29

Sona Kalanları Köpekler Isırır 30

Yapay Zeka İle Uçtan Uca Yazılım 31

Deterministik Yeti 32

AI Driven Development 33

Yapay Zeka ve Context Türleri 34

Yeni Dönemin Programcıları 36

Test Edenler Kazandı 37

Takım Olayı Bitmiştir 38

Konuyu Bilmiyorlar 39

Hangisi Daha Verimli 40

Çevik Yazılımın Rönesansı 41

Human Context Switch 42

Analizin Analizi 43

Gereksinimlere Odaklanma 44

Dağarcığın Gücü 45

Zayıf Bağlam 46

Lütfen 47

Claude Opus 48

Amiga Efekti 49

Nyet! 50

Yazılımda Döngüler 51

Bitti mi? 52

Yapay Zekaya Güvenmek 53

Frontend First 56

Yapay Zeka İle Kaybolacak Meslekler 57

Opus 4.8 ile İlk Deneyimlerim 58

AI First 59

Yazılım Hala bir Zanaat mı? 61

Full Automated Coding 62

Yapay Zekaları Birbirlerine Kırdırma 64

İpek Böceği 65

Yeni Derleyiciler LLM'ler 67

Yapay Zeka Yolculuğumun Kısa Hikayesi 68

Gereksinim Analizi 69

Artık Kod Yazmaya Gerek yok 70

Teknoloji Ötesi 72

Yazılım Artık Epistemolojidir 73

İşin Özü 74

Yeni Soyutluk Seviyesi 75

Yapay Zeka İle Nasıl Çalışıyorum 76

Gereksinim Analizi 78

Fable ile Chatgpt Nasıl Beraber Çalışırlar? 79

Fable 5 Masal Oldu 80

Loop Engineering 81

Hala Prompt Engineering mi Yapıyorsunuz? 83

Yapay Zeka Araç Kullanımı Nasıl Evrildi? 84

Intent Based Programming ve Intent Specific Language (ISL) 85

Yazar Hakkında 87

Statükocu Zihniyet

Kaynak: https://kurumsaljava.com/2025/12/21/statukocu-zihniyet/

Yazılımcı olarak bazı gerçeklerle yüzleşmemiz gerekiyor.

Copilotu sadece bir sefer Claude Sonet 4.5 ya da türevleri ile deneyimleyen bir yazılımcı, anti yapay zeka savlarının birçoğunun gerçek dışı olduğunu görecektir. Nedir bu anti yapay zeka savları?

- Vibecoding iyi netice vermez,
- Senior programcı vazgeçilmezdir,
- Yapay zeka yazılımcı mesleğini ortadan kaldıramaz,
- ve türevleri…

Yapay zekanın tecrübeli bir yazılımcıyı ortadan kaldırma ihtimalinin olmadığını düşünenlerdenim. Aynı şekilde vibecoding hızlı MVP tarzı prototipleme için kullanılabilecek bir yöntemdir. Ancak yazılıma yeni başlayanlar için vibecoding bir zehirdir. Adapte olabilen senior vazgeçilmez olacaktır.

Bunlara değindikten sonra, asıl anlatmak istediğim mevcuza gelmek istiyorum.

Statükocu zihniyet…

Bence çok hızlı bir şekilde bu statükoyu korumacı zihniyetten vazgeçmek gerekiyor. Kendimize asıl sormamız gereken soru şu:

Biz yazılımcılar için çalışma tarzı nasıl değişecek ve biz buna nasıl adapte olabiliriz?

Bugün modern bir araba fabrikasında insanlar sadece gözlemci durumda. Tamamen karanlıkta faaliyet gösteren fabrikalar var. Bir senior seviyesinde gelmiş programcının teknik olarak yapay zeka ile rekabet etmesi imkansız. Senior pogramcılar ilerde yapay zeka işini yaparken gözlemci ve kontrolcü konumunda kalabilir.

Benim yanımda örneğin benim kadar, belki de daha yetkin birisi oturuyor artık: LLM. Ben ona hergün iş yaptırıyorum ve zamandan tasarruf ediyorum. İşi bilen birisinin ona ne yapması gerektiğini anlatması işin küçük bir bölümü artık. Kısa bir zaman sonra yazılımcılar olarak çok farklı bir ortamda çalışıyor olacağız. Buna ne kadar hazırız?

Kod yazmak işin çok küçük bir kısmı. Biz yazılımcılar bu dar alanda yapay zeka ile rekabet etmek zorunda değiliz. Bırakalım angaryayı onlar yapsın. Bizi ayrıcalıklı kılacak olan hardskillller değil, softskillerdir. Onlara odaklanarak her an

artan yazılım talebini rahatlıkla karşılayabiliriz.

Yapmamız gereken tek şey değişimi kabul edip, onunla yaşama yetisi geliştirmektir.

Özcan Acar EOF (End Of Fun)

Yapay Zeka İle Çevik Olma

Kaynak: https://kurumsaljava.com/2025/12/22/yapay-zeka-ile-cevik-olma/

Yazılımda çevik olmanın tek yöntemi test yazmaktır. Yazılım projelerinin zaman içinde yeniden yapılandırılamayarak telef olmalarının tek sebebi test eksikliğidir.

Bknz: https*://kurumsaljava.com/2012/04/05/çevikliğin-boylesi/_*

Artık test yazmamak için hiçbir mazuriyet kalmamıştır. Claude Sonet 4.5 yeni bir API için 20 ye yakın entegrasyon testini 5 dakikanın altında olusurdu. O testleri yazmak günler ya da haftalar alabilir.

Test yazmamanın ana sebebi yanı maliyetli olması nedeni artık ortadan kalkmıştır.

Çevik süreçlerinin oturamamasının da ana sebebi refactoring yapma kabiliyetinin geliştirilememesi idi.

Bu taktirde XP gibi çevik süreçlerin tekrar bir rönesans yaşaması söz konusu.

Yazılımcılar İçin Yeni Dönem Başlıyor

Kaynak: *https://kurumsaljava.com/2025/12/23/yazilimcilar-icin-yeni-donem-basliyor/*

Sadece mevcut bilgi ve tecrübe seviyesini ölçmeye yönelik yazılımcı mülakatları sona erecek.

Artık adaylardan copilot gibi yapay zeka araçları ile sunulan bir fikir için çok hızlı ve çalışır bir protip (MVP) oluşturmaları istenecek. Birkaç saatlik bir zaman diliminde fikirden, çalışan ürüne kadar tüm yazılım yelpazesi ve adayın bu süreçte nelere hakim olduğu kontrol edilecek.

Böylece usta-çırak ilişkisi ile yetiştirilmeyen yazılımcılar için bunun önü açılmış oluyor.

Nasıl bir kalp cerrahi asistan doktor olarak yıllarca bir kalp operasyonunun pratik ve teorik eğitimini ustalarından alıyorsa, yazılımcıların da böyle bir ilişki içinde yetkinlik kazanmaları gerekir hale gelecek. Bunun olmadığı bir ortamda yetişen tüm yazılımcılar vibecoding ile mülakatta istenenleri yapabilir anca ne yaptıklarını açıklayamayacak halde olacaklar.

Artan yazılım talebi ile yazılımcılar bu işin kendi kendilerine öğrenmeye bırakılamayacak kadar kritik unsur haline geliyorlar. Yazılım bir zanaattır ve ustası tarafından öğretilmek zorunda.

Bknz: `https://kurumsaljava.com/2012/05/08/programcilik-sanat-mi-zanaat-mi/`

Yazılımcılar İçin Yapay Zeka Kullanma Kılavuzu

Kaynak: https://kurumsaljava.com/2025/12/29/yazilimcilar-icin-yapay-zeka-kullanma-klavuzu/

Ben günlük islerimde IntelliJ / Android Studio ve Copilot Claude Sonnet 4.5 yapay zeka modelini kullanıyorum.

Zaman içinde kendim için yapay zeka öncesinden çok farklı bir çalışma modeli gelistirdim. Bu bir nevi yapay zeka kullanim klavuzu. Yapay zeka araçları gelistirildi lakin bunlarla nasıl programcı olarak calismamiz gerektiğine dair bize bir kullanim klavuzu verilmedi. Herkes kendi başına bunları kesfetmek zorunda. Bu konuya katki amacıyla kendi tecrübelerimi paylaşmak istedim. Hep birlikte belki genel kapsamli bir çalışma ve kullanim klavuvuzu gelistirebiliriz. Bu yazım benim için bir nevi "programming best practices with ai" görevini görecek.

Yeni Soyutluk Seviyesi: Eskiden programcı olarak mimari, nesneye yönelik programlama, tasarim sablonlari gibi konularla istigal ederek, program gelistirirdik. Şimdilerde daha yüksek bir soyutluk seviyesinde, yapay zeka ile gelen prompt engineering, context, mcp, token ve agent gibi yeni terminoloji, yöntem ve araçları kullanarak program yazıyoruz. Bu ister istemez program yazma tekniklerimizi, aliskanliklarimizi ve soyutlama yetimizi dogrudan etkiliyor. Burada atabilecegimiz en doğru adim soguk suya atlamak yanı biran önce bu yeniliklere adapte olup, gerektiği şekilde hareket edebilmek olacaktır.

Doğru Model: Ben gpt 5 ve türevleri ile çok sağlıklı sonuç alamadim. Gemini türü modelleri de denedim. Lakin Claude Sonnet 4.5 ile en iyi sonuclari aldim. Bu konuda herkesin LLM secimi yapılan işin baglamina göre degisecektir. Doğru modelin çalışma verimliligini artırır, yanlış model ile çalışmak günün sonunda hiçbir şey basaramamislik hissi birakabilir.

Copilot Ayarlari: Copilot için proje workspace içinde copilot-instructions.md isimli dosyada yapay zekanın islem yapmadan önce dikkate almasi gereken hususlari listeledim. Burada metotlarin nasıl yapilandirilmasi gerektiginden, UI tasarimi, kullanilacak çatılar, kod stili ve testlere kadar birçok konuda yapay zekaya yardim edici bilgiler vermek mümkün. Buradaki amac çıktının daha deterministik ve belli bir yapıda olmasini sağlamak.

Prompt Mühendisligi: Bir yazılım mühendisi artık bir prompt mühendisine dönüsmek zorunda. Ben bu tabiri çok hos bulmuyorum, lakin programcının artık kendi anadilinde, gelistirilmesini istediği uygulamanın nasıl olması gerektiğine dair en ince detaya kadar bunu net bir şekilde ifade edebilir ve kagida dökebilir hale gelmesi gerektiği zorunlulugunun olduğunu düşünüyorum. Yapay zeka ile interaksiyonda destan yazmaya gerek yok. Prompt ne kadar sade ise, o oran-

da çıktının beklentileri karsiladigini gördüm. Buradaki en önemli nokta context olarak ifade edilen kapsama alaninin iyi tanimlanmis ve doğru verilerle doldurulmus olmasıdır. Ben örneğin copilot ile çalışırken her prompt için gerekli sınıfları, görselleri ve belgeleri context içinde tanımlıyorum. Bu şekilde copilot ve dolaylı olarak Claude Sonnet benim prompt içinde ifade ettiklerimi workspace içinde yer alan icerikle daha iyi eslestirip, yapılması gerekenleri tespit ettikten sonra, hayata gecirebiliyor.

Kısaca ne kadar prompt, o kadar köfte!

Prompt History: Ben workspace içinde prompt.md isminde bir dosyada, kullandığım tüm promptları tutuyorum. Aslında güncel prompt için aklima gelen ilk düsünceleri bu dosyaya eklemeye başlıyorum. Ilk düzeltmelerden sonra promptu en güncel hali ile copilota veriyorum. Zaman içinde bu bir nevi dokümentasyona dönüşüyor ve günler sonra tekrar yazdiklarima göz atarak, nasıl buraya geldigime dair sağlam bir fikre sahip olabiliyorum. Bazen copilot ile oluşan kodu revert / rollback yapıyorum. Bu gibi durumlarda da aynı promptu history dosyasinda lokalize ederek, yeniden kullanma sansim oluyor.

Architecture First : Hiçbir ön hazırlık yapmadan copilota kod yazdirmanin ismi vide coding. Bunun nereye varacagi malum. Bu sebeple yeni bir uygulamaya basladigimda kesinlikle yapay zeka araci kullanmiyorum. Uygulamanın temellerini kendi bilgi, tecrübem ve düsüncelerim dogrultunda elimle atmam gerekiyor. Bu şekilde ilk mimari ve katmansal yapı ortaya çıkıyor. Akabinde yapay zeka oluşan yapiyi kendi isinde kopyalarak, gerekli kod parcalarini oluşturması daha kolay hale geliyor.

Building Blocks & Subprompting: Ben bir uygulama özelliğini (feature) yapay zekaya bir prompt ile yaptirmaya calismiyorum. Bu çok iyi sonuclar veren bir yöntem olmadi benim için. Bunun yerine uygulama özelliğini küçük parcaciklara bölerek, her parca için bir prompt yaziyorum. Burada öncelik her zaman uygulama özelliginin temelini olusturan domain sınıfları, veri tabani yapısı, repository ve servis sınıfları gibi yapıları ortaya cikartacak temel promptlar aracılığı ile uygulama özelliginin temellerini atabilmek. Temeller insa edildikten sonra yeni promplar ile kat cikilmasi yanı istenilen türde bir uygulama özelliginin programlanmasi kolaylasiyor.

Gereksiz Muhabbet: Copilot ile gereksiz yere “iyi yaptin aferin” yada “tesekkür ederim” gibi şeyler yazarak sohbet etmiyorum. Bu tür geri bildirimlerin programcı olarak size hiçbir faydasi yok. Yapay zeka bu sebeple gereksiz yere enerji sarf eder ve siz de en kötüsü faydasiz bir islem için token harcamis olursunuz.

Sürekli Commit: LLM ile çalışmak klasik programlama mantığına aykiri bir durum. Programcılık mantık gereği deterministik ve yer yer idempotent olmak zo-

runda. Lakin ben aynı promptun aynı sonucu verdigine hiç sahit olmadim. LLM'ler deterministik yapılar değil, istatistiksel yapılar, yanı çıkan sonuclar da ne yazık ki istatistiksel olabilirlik hesaplamalarina dayaniyor yanı deterministik olamiyorlar.

Bunu bildigim için her prompt öncesinde son yapilanlari korumak amacıyla git commit ile tüm değişiklikleri saglama alıyorum. Prompt sonrasında oluşan program parçaları hosuma gitmiyorsa, rollback ile cikis noktama geri dönüyorum ve promptu yeniden sekillendirip, devam ediyorum. Sonuç hosuma gittigi taktirde code review yaparak tekrar bir commit alıyorum ve bir sonraki prompta geciyorum.

Olmadiysa Rollback: Biraz önce bahsettiğim gibi LLM çıktıları deterministik değil. Bunun yanı sira prompt ve context yetersiz kaldigi için cikti istenilen türde olmayabilir. Bu durumda kodu degistirmek yerine rollback yaparak, sıfırdan başlıyorum. Kodu degistirmek ve calistirmaya çalışmak çok zaman kaybina sebep olabilir. Bu yüzden sıfırdan baslayabilek her daim daha iyi bir opsiyondur.

Regresion Testing: Eğer promptlar ile test yazmiyorsaniz ya da yazdirmiyorsaniz, geçmiş olsun. Er ya da gec kullandiginiz LLM daha önce yaptığı değişiklikleri ezecek ve çalışır durumda olan program parçaları çalışmaz hale gelecektir. Buna programcılar arasında regresyon – geriye gitme denir. Bunun için özel bir test kategorisi bile vardır: regresyon testleri.

Ne kadar yapay zeka da dense, LLM bir yapay zeka olmakdan ışık yili kadar uzak. Bu yüzden geniş kapsamli degisikliklerde uygulamanın bazı bölümlerinde istenmeyen değişiklikler vücut bulmus olabilir. Bunları tespit etmenin tek yolu, otomatik çalışan testlerdir.

LLM'i sürekli refactoring yapan bir makina gibi görmek gerekiyor. Her prompt potansiyel bir refactoring. LLM istediği şekilde kodu yogurur. Bunun mutlaka ve mutlaka yan etkileri olacaktır. Bunları tespit etmenin tek yolu test yapmaktir.

Sağlam bir test seti olmayan programcının promptlar ile gelistiridigi bir uygulama canlıya alindiginda en iyi ihtimalle deterministik olmayan davranislar sergileyebilirken, en kötü ihtimalle çalışmaz hale gelebilir. Bu sebeple her prompt mutlaka gerekli test setinin olusmasini mecbur kilmalidir.

Continuous Review: LLM ciktisi bir kodun programcı tarafından iki sebeple gözden gecirilmesi yanı code review yapılması gerekmektedir.

Ilk sebep kodun ne kadar geçerli oldugunun kontrolüdür. Kod genel geçerli kurallara uymadigi için yeniden yapilandirilabilir. Bunun için mutlaka bir code review yapılması zorunludur.

Ikinci sebep benim için daha önemli bir sebep. Ben LLM ciktisi kodu okuyarak

çok şey ögreniyorum. LLM tarafından oluşturulan kodun, bu gezegen üstünde yayasan tüm programcıların açık kaynak olarak github ve co üzerinde tuttuklari kodlarin damitilmis hali olduğunu düsünecek olursak, her ciktidan çok şey ögrenmenin ne kadar enfes olacagi asikardir.

Her dönemi kendine has (bknz. Kant – Zeitgeist) bir ruhu, yasam ve çalışma şekli var. Programcılar olarak yeni bir döneme girdik. Hızlı adapte olanlar ayakta kalirken, digerlerine ne olacak bilinmez.

Özcan Acar EOF (End Of Fun)

Code Generation ve Generative AI

Kaynak: https://kurumsaljava.com/2026/01/03/code-generation-ve-generative-ai/

Code generation konusunda nereden nereye geldik...

- Üniversite yillarinda CASE araçları ile UML modelleri hazirlar ve tüm interface, dto ve entity gibi siniflarin otomatik olarak olusturulmasini saglardik.
- Corba ve IDL ile servis arayüzlerini (interface) tanimlardik, ORB üzerinden remote object ile iletişim için gerekli lokal proxy stub kodu otomatik olusturulurdu.
- 2000 öncesi EJB döneminde local ya da remote interface sınıfları olustururduk, implementasyonlarini EJB compiler/container kendisi yazip, kosturur-du.
- AOP (Aspect Oriented Programming) ile aspektler yazardik, gerekli baglantilari oluşturmak için aop derleyicisi kod olusturdu ve derledi.
- Spring ile tüm konfigürasyonu xml dosyalarinda olustururduk, proxy implementasyonlarini ve enjeksiyonu Spring IOC container yapardi. Daha sonra anotasyonlar kullanmaya basladik, arka planda gerekli kod yine build ya da runtime içinde olusturulurdu.
- Mapstruct ile entity – dto mapping için gerekli kodu yazmaya gerek kalmadı.
- Lombok ile tüm get/set metotlarindan kurtulduk.

Kodu generate ettirmek için kod yazarken, şimdilerde kod yazmadan gen ai ile tüm kodun olusmasini seyrediyoruz.

Bundan sonra ne gelecek?

Makina kodu yazmanin ne kadar zor olduğunu hatirlayalim. Assembly ile programcilikta büyük bir sicrama yasandi. Bu da yetmedi ve Fortran, C ve Java gibi diller ile daha yüksek bir soyutluk seviyesine cikildi.

Böyle bir sürecin aynisi yapay zeka kullanımı için yasanabilir. Artık yapay zekanın ürettigi koda hakim olamayan programcılar için yeni araçlar gelistirilecek. Onlar yardımı ile yapay zeka kullanımı daha da soyutlanacak ve kontrol edilebilir hale gelecek.

Sistemsel olarak imperatif programlama ile yapay zekaya kodu nasıl yazilacagini söylemek aynı şeyler. Bu detaylarda kaybolmak anlamına geliyor. Oysaki deklaratif tarzda ne yapılacağını söylememiz yeterli. Gerisini kullandigimiz programlama dili, araç ya da yapay zeka halletmeli. Gelecekte daha çok imperatif araçlar kullanarak kod yaziyor olacağız.

Mindshift

Kaynak: `https://kurumsaljava.com/2026/01/03/mindshift/`

Benim 20 sene imperatif program kod yazdiktan sonra fonksiyonel proglamaya gecisim çok zor olmustu. Beynimin en son hücresi bile imperatif düsünmeyi yeglerken, artık bunu başka türlü yapıyoruz demek yeterli olmuyor. İnsanın bu kemiklesmis düşünce sablonlarini asmasi çok zor.

Yapay zeka baglaminda günümüzde senior (10+ yil tecrübe) olan programcıların da buna benzer bir problem ile karşı karsiya olduklarini düşünüyorum.

Vibe coding yapan ama yapay zeka araçlarını kullanmayı ögrenmis junior programcıların bahsi geçen eski dönem programcilarini solladiklarini görebiliriz.

Bu sebeple degisime karşı koymadan hızlı bir şekilde adapte olmak gerekiyor. Aksi taktirde programciliktan ekmek yemek artık mümkün olmayabilir. Bunun ilk ibareleri uzun yıllar programcılık yapmis senior programcıların sektör degistirme ya da yönetim kademesine gecmeleri ile kendisini gösterecektir.

Analog Yazilimdan Dijital Yazılıma Geçiş

Kaynak: https://kurumsaljava.com/2026/01/05/analog-yazilimdan-dijital-yazilima-gecis/

Yazılım camiasindaki güncel gelismeler Alice harikarlar diyarında gibi hissettiriyor.

Artık iki dünya olustu: analog yazılım, dijital yazılım.

Benim gibi uzun yillardan beri yazılım yapanlar analog yazılım dünyasinda yetistiler.

Bugün yazılıma yeni basliyanlar, dijital yazılım dünyasinin parcasi olarak yetisecekler. Onların mevcut yapılara adapte olmaları çok daha kolay.

Birkaç ay öncesine kadar "vibe coding" in yeni başlayanlar için uygun bir yöntem olmadığı fikrine sahiptim. Bu fikrim son günlerde değişmiş durumda!

Genç arkadaşlar: hiç vakit kaybetmeyin ve vibe coding yapmaya baslayin. Sizler dijital native olarak bu yeni yazılım dünyasina dogdunuz ve orada yetiseceksiniz. Yaninizda artık tam tesekküllü ve dünyanin en donanimli yazılımcı ustası oturuyor. Size yazılımın nasıl yapılacağını en kısa sürede ögretecek. Sürekli ona kod yazdirin ve anlamadiginiz yerleri sorun. Bizim yıllar boyu calisarak ögrendigimiz konuları siz ışık hizinda ögreniyor olacaksiniz.

5 yil içinde çok daha iyi programcıların yetistigini görecegiz.

Bir girisimci olarak yazılım ekibimde yapay zeka ile çalışma süreclerine hakim junior yazılımcılara öncelik verecegim. Onların yetistirilmesi ve konuya hakimiyetleri çok daha kolay olacak.

Senior programcılar analog yazilimdan dijital yazılıma geçiş sürecinde avantajli konumdalar eğer bu değişimi kabul edip, yeni süreclere adapta olabilirlerse.

1990 yilinda kod yazmaya baslamis ve son 30 yildir profesyonel yazılımcı olarak çalışan birisi olarak sunun altini cizerek söylemek isterim:

"Yazılım konusunda ben bundan daha heyecanli bir gelisme yasamadim"

Junior Yazılımcılar ve Vibe Coding

Kaynak: https://kurumsaljava.com/2026/01/06/junior-yazilimcilar-ve-vide-coding/

“Vibe coding” terimi mevcut durumu tanımlamak için artık yetersizdir.

Genç yazılımcılar artık vibe coding yapmiyorlar, usta çırak ilişkisi içinde gerekli tüm temel ve üst bilgiyi hocalarindan (AI) ögreniyorlar.

Bizim dönemimizde eksik olan usta-çırak ilişkisi artık olusmus durumda. Genç yazılımcıları bu nimetten faydalanmalari yerine uyarmak anlamsız. Yaptiklari vibe coding değil, yeni düzenin ögrenme süreci ve bu sürecin sonunda kendilerini iyi yetistirmis olacaklar.

Neden bu kadar genç yazılımcılara güveniyorum? Çünkü bir yazılımcı meraklidir. Meraklı olmayan birisi yazılımcı olamaz. Bu merak onlari yapay zeka tarafından oluşturulan kodun nasıl calistigini anlamaya istecek ve bu şekilde ögrenecekler. Imtihan için kopya hazirlayan ögrencileri düsünün. Onların kopya cekme tesebbüsleri zaten gereken seyleri ögrenmelerini sagladi.

Bu sürece yeni bir isim konmasi gerekiyor. Genç yazılımcılara sadece vibe coding yapıyorlar demek artık doğru değil.

Peki vibe coding nedir? Hiçbir yazılımcı egitimi ve altyapısı olmayan ve daha önce kod yazmamis birisinin yapay zekaya kod yazdirip, programcı olduğunu düsünmeye baslamasidir. Burada kısa vadece prototip ve mvp tarzı ürünler cikabilir lakin sürdürülebilirligi yoktur, çünkü yazılım sadece kod yazmaktan ibaret değildir ve ustalik gerektirir.

Sizce junior programcılar için vibe coding yerine hangi terimi kullanabiliriz? Önerilerinizi bekliyorum.

Kim Code Review Yapar?

Kaynak: https://kurumsaljava.com/2026/01/07/kim-code-review-yapar/

Copilot / Sonet 4.5 sadece fkir olarak tanimladigim uygulama için geniş kapsamli feature listesi oluşturup, bunları 45 dakika içinde 6.319 satır kod ve 46 Dart dosyası olarak implemente etti. Aynı zaman zarfinde uygulamanın ihtiyaç duydugu backend API de olustu.

Bu ufak çaplı bir uygulama idi. Geniş çaplı bir uygulamada kod satır adedi 100K ya ulasabilir.

Soru şu: hangi zaman diliminde kac programcı bu kadar kodu review edebilir?

Oluşan darbogaz görülüyor mu? Bu sadece code review kısmı. Bunun gibi kurumsal projelerde asilmasi gereken birçok teknik ve bürokratik engel mevcut.

Artık yazılımcı ekipleri günler içinde oluşan milyonlarca satır yeni kod ile bogusuyor olacaklar.

İşin altindan kalkmanin tek yolu, oluşan kodu olduğu gibi kabullenip, onay/kabul testleri ile uygulama davranislarini test etmekten geçiyor. Uygulama istenilen davranış bicimini sergiledigi sürece kodun nasıl implemente edildigini önemli değil. Uygulama ve dolaylı olarak yapay zekanın oluşturduğu kod blackbox mahiyetinde ve her daim degistirilebilir olma statüsünü tasimaktadir.

LLM'ler Çağımızın Yeni Derleyicileri

Kaynak: https://kurumsaljava.com/2026/01/07/llmler-cagimizin-yeni-derleyicileri/

Şu anda LLM ler turing complete değil. Er ya da gec LLM ler derleyicilerde olduğu gibi deterministik sonuclar üretmeyi ögrenecekler.

Assembly kodunun elden yazildigi günler bunun için güzel bir örnek. Yüksek dil derleyicileri assembly kodu / microcode üretmeye basladiklarinda, assembly programcıları bu ciktilara çok güvenmez ve ciktiklari elden kontrol ederlermis. Bunu artık kimse yapmıyor, çünkü derleyicilerin %100 doğru calistiklarindan eminiz. Hata derleyici de değil, yazılımcının yazdigi koddadir.

LLM ler çağımızın yeni derleyicileri ve çıktıları kısa bir zaman sonra kimse kontrol etmeyecek ve olduğu gibi doğru olarak kabul edilecek. Çıktının çalışmıyor olmasinin tek sorumlusu yine o cikti için gerekli olan bilgiyi (prompt, spec, context) yanlış tanimlayan programcısı olacak.

Test Yazmaya Gerek Kalmadı

Kaynak: https://kurumsaljava.com/2026/01/09/test-yazmaya-gerek-kalmadi/

Artık kodu yapay zeka yaziyor, bunu kabul ettik. Peki biz yazılımcı olarak işin neresindeyiz? Bunu burada uzun uzun anlatmak istemiyorum. Yazılımcı olarak isimiz başka alanlara kaymis durumda, ama artık kod yazmayacagiz.

Artık en önemli görevlerimizden birisi test yazmak / yazdirmak ve yapay zekanın oluşturduğu kodun yanı oluşturmak istediğimiz ürünün davranislarini test etmek. Ama test kodunu da tamamen yapay zekaya bırakmak, onun calip, onun oynadigi bizim ise sadece seyredici oldugumuz bir ortam dogurur. Bu da istediğimiz bir durum olamaz. Bizim girdileri ve çıktıları kontrol edebilecegimiz yeni bir soyutluk seviyesine ihtiyacımız var.

Cogumuz "yapay zekaya testleri de yazdirabiliriz, hem de bunu bizden daha geniş kapsamli yapar" diyecektir. Burada bazı problemler görüyorum:

- Kodu degistiren yapay zekanın testleri de adapte etmesi tehlikeli ve bu kontrol edilemez sonuclar dogurabilir (kendi calip, kendisi oynar metaforu)
- Statik test kodu sürekli kirilmaya müsait oldugundan, bakimi çok zordur. Yine burada "bunu yapay zeka halleder" demek uygun değil, çünkü elimizde tutmak istediğimiz bir kontrol mekanizmasina ihtiyacımız var
- Test etmek istiyoruz lakin kod yazmak, bakimini yapmak ve adapte gibi detaylar ile ugrasiyoruz. Bunun yerine nasıl yerine ne sorusunu sormamız gerekmektedir. Neyi test ediyoruz sorusu test etmek istediğimiz isletme mantığına odaklanmamizi saglayacaktir. Burada imperatif düşünce yerine neyin test edildigini tayin ettigimiz deklaratif düşünce tarzina geçmiş oluyoruz.

Ne yapabiliriz?

Artık düsüncelerimizi belli kaliplara sokarak bilgisayar ile çalışma zorunlulugumuz ortadan kalkmis durumda. Biz yapay zeka ile kendi dilimizde iletişim kuruyoruz. Çıktıları kontrol etmek için kullanacagimiz yeni soyutluk seviyesi de bu.

Bir uygulamanın feature olarak tanimladigimiz ve implemente ettigimiz değişik türde davranislari olabilir. Bu davranislarin uygulama tarafından sergilenebilmesi için kullanicilar tarafından tetiklenmeleri gerekir. Tanimli bir input gerekli davranış bicimi uygulandiktan sonra tanimli bir output doguracaktir. Programcı olarak bu girdi ve çıktıları kendi dilimizde tanimlayabilir ve yapay zeka ya da test çatısı tarafından uygulamaya karşı kosturulmalarini saglayabiliriz.

Kendimden bir örnek vereyim. Ben testleri artık bir yml dosyasinda şu şekilde specler olarak tanımlıyorum:

```
name: 'Clean Start Test'
description: 'When SQLite-Database deleted, then app starts in default setup mode'

tags:
  - clean

setup:
  - 'Delete database'
tests:
  - name: 'App starts with setup screen'
    steps:
      - 'Start the app'
      - 'Wait 2 secs '
      - 'Expect that "Lütfen cihaz türünü seciniz" iş visible'
```

Sonnet'e bu specleri okuyup (parse, analyse), teste dönüstürüp (generation), kosturabilen (execution) ufak çaplı bir framework yazdirdim.

Bu specler tanimlamis olduğum onay/kabul kriterlerini ihtiva ediyor. Bir nevi BDD (behaviour driven development) yapıyorum, ama onun ismi artık BDD değil, BDC (behaviour driven check). Testleri bu şekilde tanimladiktan sonra, kosturuyorum. Sonnet'in oluşturduğu test framework testleri kosturmadan önce spec dosyalarını yapısal analiz yapiyor ve akabinde test kodunu olusturarak, bunları koşturuyor.

Ben bir flutter uygulamasını test etmek için bu yöntemi sectim. Bir web uygulamasını bu şekilde tamamen yapay zeka yardımı ile test etmek çok daha kolay. Playwright isminde bir MCP server mevcut. Onun yardımı ile Sonnet gibi bir LLM spec dosyası içinde bulunan test adimlarini MCP aracılığı ile bir web tarayicisina baglanarak kosturabilir.

Kısaca özetleyecek olursak:

- Artık test kodu yazmıyorum ya da bakmini yapmiyorum.
- Testlerim kirilmiyor.
- Uygulamanın davranislarina odaklaniyorum ve neyin test edilmesini gerektiğini tayin ediyorum.
- Yapay zekanın sadece kodu adapte etmesine izin veriyorum, test kodunu (specler) değil. Bu şekilde neyin test edilecegi hakimiyeti bende kaliyor.
- Yapay zekadan destek alarak geniş kapsamli specler olusturabiliyorum.

Bu speclerin olusturulma adimini da şu şekilde otomatize edeceğim. Jira bünyesinde yeni bir feature tanimlarken onay/kabul kriterlerinin de tanimlanmasi gerekiyor. Ben bir kodlama ajanina bu feature i implemente etmesini söylediğimde, ajan ilk adim olarak gerekli test spec dosyalarını. oluşturacak. Ajan işini tamamladiktan sonra test için oluşturduğu spec dosyalarını derleyerek, koşturuyor olacak. Bir hata ile karsilasmasi durumunda implementasyonun istenen seviyede olmadığını anlayarak, gerekli adaptasyonlari yapmaya başlayacak. Böyle bir döngü ile aslında hiç insan interaksiyonu olmadan bir uygulamayı feature by feature implemente ettirmek mümkün.

Test ekibine hos geldin yapay zeka :)

Yapay Zeka ile Otomasyon Çılgınlığı

Kaynak: https://kurumsaljava.com/2026/01/10/yazay-zeka-ile-otomasyon-cilginligi/

Artık nereye bakarsaniz claude code ya da türevleri ile yazılım projelerinde her şeyi otomatize etmeye calisanlari göreceksiniz. Kesinlikle bir satır kod yazma niyetleri olmadığı gibi gereksinim analizinden dogan artefaktlari yapay zekanın alabilecegi bir yerde (örng. backlog) tutup, gerisini full otomatize etmeyen çalışan tiplerden bahsediyorum. Genelde bu tipler "aksamdan bir dügmeye bastim, sabah kalktigimda tüm uygulama hazirdi" minvalinde söylemler paylasirlar.

İşte bunlar vide coding yapan sahislar arkadaşlar. Gerçek anlamda yazılım ile ugrasan birisinin izleyebilecegi bir yol değil anlatilanlar. Yazılım gerçekten tecrübe gerektiren birşey ve bahsedildigi şekilde otomatize edilmesi mümkün değil. Bunun için gereksinim analizlerinin %100 gercekleri yansitiyor ve en küçük birimlerine kadar parcalanmis olması gerekiyor. Bu size birşey hatirlatti mi? Evet waterfall!

Evet, ben tüm kodu artık yapay zekaya yazdiriyorum, ama bir prompt ve dolaylı olarak bir gereksinim adam akilli tanimlanmamis olsun! Bu durumda kesinlikle kabul edebilecegim bir sonuç ortaya cikmiyor. Kodu silip, prompt refactoring yaparak sıfırdan almam gerekiyor.

Kod yazma derdinden kurtulduk diyelim, ama bu bizim başka bir alanda çok daha iyi olmamizi gerektiriyor: gereksinim analizi, onun makul parçalara bölünmesi (user story), onay/kabul kriterlerinin tanimlanmasi, yapay zekanın promptlar aracılığı ile yönlendirilmesi ve işini bitirdikten sonra review.

Yapay Zeka İle Deklaratif Programlama Çağı

Kaynak: https://kurumsaljava.com/2026/01/27/yapay-zeka-ile-deklaratif-programlama-cagi/

Promptlar ile kod yazdirmak deklaratif programlama tarzidir. Nasıl değil ne yapılacağını beyan etmemiz yeterlidir.

Deklaratif apiler araciligiyla neyin nasıl yapıldığını bilmeden zaten yillardir deklaratif programliyorduk. Şimdi nasıl kismini LLMlerin üstlenmis olması bizim çalışma ve düsünme tarzimizda birşey degistirmez.

Bugünün programciligi ehliyeti olan herkesin istediği marka bir araci sürebiliyor olması ile aynı seydir. Ehliyetim var ama 3 silindirli ya da icten yanmali bir motoru süremem demek ile yapay zekanın yazdigi koda güvenmiyorum demek aynı seylerdir.

Programcilikta müsterinin istekleri esastir. Yapay zeka devrimi ile tamamen buna odaklanabilecegiz büyük bir firsat elimize geçmiş bulunuyor. Programciligin gelecegi bundan sonra çok daha parlak ve çok daha eglenceli.

Tuvaldeki Uygulama

Kaynak: https://kurumsaljava.com/2026/01/29/yazilimci-kariyerim/

Degerli arkadaşlar,

yapay zeka ile yazılımın nereye doğru gittigini size resmetmeye çalışacağım. Kendim, biraz sonra okuyacaklarinizi intensif bir şekilde son günlerde yasiyorum, yanı dogrudan tecrübelerimle sabit.

Ben sadece 3 gün içinde bir device management sistemi gelistirdim. Bu cihaz tarafında go ile bir agent, sunucu tarafında bir spring boot + angular uygulamasi. Iki taraf içinde hiç kod yazmadım. Sistem düşündüğümden çok daha iyi tasarlandi ve şu anda çalışıyor.

Önümüzdeki zamanlarda yazılım artık asagida anlatmaya çalışacağım şekilde yapiliyor olacak. Size düsen yazılımcı olarak bu işin neresinde oldugunuzu düsünmeniz.

Şimdi beyaz bu tuval hayal edin. Bir ressam hayal ettiği seylere bu tuval üzerinde hayat verir.

Bizler de yazılımcı olarak buna benzer birşey yapıyoruz. Aramizdaki tek fark, bizlerin müsterilerimizin istekleri doğrultusunda o tuvali sekillendirmemiz.

Şimdi bu tuvali bir yazılımcı perspektifinden boş bir web sayfasi, mobil ya da desktop uygulama ekrani olarak düsünün.

Şimdi tekrar ressama geri dönelim. Ressam her fircasi ile tuval üzerinde hayal ettiği resmi olusturmaya baslar. Aynı şeyi bir uygulama yaptirmak isteyen herhangi bir şahıs için gözümüzde canlandiralim. Bu noktadan itibaren anlatacaklarim, gelecekte yazılım ürünlerinin nasıl gelistirilecegini gösteriyor olacak.

Şahıs boş ekranin karsisinda oturuyor ve yapay zekaya sesli olarak şu komutlari veriyor:

=== Şahıs === Sol tarafta bir sidenav menü istiyorum. İçindeki şöyle şöyle menü elementleri mevcut.

=== Yapay Zeka === Calismaya başlıyor ve menüyü olusturdugunu söylüyor. Müsteri menüyü aninda o boş ekranda görüyor.

=== Şahıs === Şimdi orta bölümde müsteri listesini görmek istiyorum. Ayrıca müsterileri sana verecegim kriteler doğrultusunda filtreleyebilmeliyim.

=== Yapay Zeka === Gerekli tüm customer entity, repository, service, controller, liquibase script, angular component ve diğer yapıları olustuyor ve şahıs talep

ettiği müsteri listesinde ekranda görmeye başlıyor.

Müsteri taleplerini arka arkaya siraliyor ve bir ressamin tuval üzerinde resim yaparcasina uygulama yavaş yavaş ekranda belirmeye başlıyor. Bu sadece bir mock, mvp ya da prototip değil, tam tesekküllü bir uygulama.

Benim son üc günüm hemen hemen bu anlattigim şekilde gecti. Claude opus 4.5 kullanıyorum. Şu an bu benim isimi mükemmel bir şekilde gören bir LLM. Daha ileri versiyonlari programcılara ihtiyaç duyulmadan benim resmetmeye çalıştığım şekilde çalışıyor olacak. Herhangi bir uygulamaya ihtiyaci olan birisi ekran başına oturacak ve ne istedigini sadece söyleyecek ve gerisi bir sihibazin isiymis gibi ekranda belirmeye başlayacak.

Prompt Engineering

Kaynak: https://kurumsaljava.com/2026/02/27/prompt-engineering/

Herkes prompt yazarken sade ve detaylı bir dilin yeterli olduğunu düsünüyor. Prompt yazarken kullanilan dil ilk etapta çok önemli değil, yanı gramatiksel kusursuz cümlelerin yazilmasi gerekmiyor. Daha önemli olan şey bağlam yanı context. Context yapılacak iş hakkında plain text haricinde resim, url, kod, video, teknik terimler vs gibi yapılacak ise işaret eden yapılar ihtiya etmek zorunda.

LLM'i karanlik odada duran bir cisim gibi düsünün. Size doğru sonuclar verebilmesi için o odayi aydinlatmaniz gerekiyor. Bu sadece context üzerinden ve onun nasıl yapilandirildigi ile ilgili bir durum. Ona siir okuyarak nereye varabilecegimiz asikar. Ama devrik cümlelerle bile yarim yamalak teknik çerçeveyi tanımlamak yeterli olabilir.

Bu yüzden prompt engineering denilen şeyi en iyi yine yazılımcılar yaparlar, çünkü yapılan işi teknik olarak en iyi onlar ifade edebilirler. Vibe coding yapanlari üzdügüm için tekrar kusuruma bakmasinlar. Yazılımcı kimligine bürünemeden ürün cikarmaya çalışmak abesle istigaldir.

Yapay Zekanın Maliyeti

Kaynak: https://kurumsaljava.com/2026/03/16/yapay-zekanin-maliyeti/

Geçen ay Github Copilot için 173 dolar ödeme yapmisim. Sürekli Claude Opus 4.6 kullandim. Yazilan kod 5 kisilik senior bir ekibin belki 2 ayda cikacagi türdendi. Bir senior yazılımcının tüm giderleri ile maliyeti 200K TL civarinda. Bu durumda 2 milyon TL gider karsisinda 173 dolar duruyor. Bir girisimci için bundan daha güzel bir gider azaltma kalemi , ama bir yazılımcı için bundan daha hüzünlü bir tablo olamaz, çünkü 5 kisilik bir senior ekibin çalışma dinamigi, ruhu, pratigi ve eglencesinin parcasi olmanın karsiligini para ile ölcmek imkansız.

Yeni düzen bu. Ne yapacagiz? Alisacagiz.

Sona Kalanları Köpekler Isırır

Kaynak: https://kurumsaljava.com/2026/03/17/sona-kalanlari-kopekler-isirir/

Yazılımcılar yapay zeka konusunda bu aralar üc gruba ayrilmis duruma:

- Ilk deneyimini yaptiktan sonra bir ise yaramadigini düsünenler
- Hiç ilk tecrübesi olmayanlar
- Yazillimi tamamen yapay zekaya devredenler

Ilk iki kesim genelde yapay zeka hakkında yapılan olumlu yorumlara kendi olmayan tecrübeleri isiginda karşı koyuyurlar, yanı ücüncü kesim bunları ikna etmekle meşgul.

Soru şu:

Bir devrim tamamlandiginda, doğru tarafta olmayanlarin başına ne gelir?

Almanca'da bir deyim var: "die letzten beißen die Hunde" (sona kalanları köpekler ısırır).

Nacizane, sona kalmayin derim.

Yapay Zeka İle Uçtan Uca Yazılım

Kaynak: https://kurumsaljava.com/2026/03/17/yapay-zeka-ile-uctan-uca-yazilim/

Yapay zeka konusunda kemiklesmis, bilgi sahibi olmadan fikir beyan edisler artmis durumda. Burada konuyu biraz daha aydinlatabilmek için kendi düsüncelerimi ve bu konudaki tecrübelerimi paylaşmak istiyorum.

Ben yapay zeka araclarina hakimiyetin bastan sona ve uçtan uca projeler gelistirildiginde olusmaya basladigini tecrübe ettim. Geçen seninin aralik ayindan itibaren Claude Sonet 4.5 i kesfetmem ile yazilimi tamamen yapay zekay birakmis durumdayim. Üzerinde çalıştığım proje çok geniş kapsamli bir platform. Bir platforma dösünebilmesinde en büyük etken yine yapay zeka oldu, çünkü eskinden zamansizliktan dolayı hiç baslayamayacagim alt projeleri kısa sürede yapay zeka yardımı ile tamamlayarak, istediğim platformu olusturmam daha kolay bir hale geldi. Bu proje bünyesinde en uçtan (web, mobil) en dibe (backend) kadar her türlü yazilimi yapay zeka ile birlikte yapıyoruz. Bunun şöyle bir avantaji var. Client neye ihtiyaci olduğunu beyan etmesiyle birlikte ben paralelde başka bir ajan ile backend kismndaki gerekli değişiklikleri yapıyorum. Yanı backend içinde kafama göre birşeyler implemente etmiyorum ya da ettirmiyorum. Oluşan kodun tamamini client talep ediyor. Uçtan uca bu şekilde hareket ediyor olmak yapay zekanın doğru kullanimini kolaylastiriyor, çünkü ben tamamen gereksinimlere odaklaniyorum ve yapay zekayı yönlendirmem yeterli oluyor.

Eğer yapay zeka ile çalışma yöntemini tam anlamıyla kavramak ve benimsemek istiyorsaniz, uçtan uca bir proje yapmak zorundasiniz. Hello world vari backend api oluşturmak ya da bir websayfasi tasarlatmak ile yapay zeka araclarinin özlerinde nelere kabil olduklarini anlamak imkansız. Yapay zeka araclarinin iyi kullanan bir yazılımcının yaptığı şey aslında gereksinimlere odaklanmak ve teknik olarak yapay zekayı gütmektir. Gerisi zaten corap sökügü gibi gelir.

Deterministik Yeti

Kaynak: https://kurumsaljava.com/2026/03/18/deterministik-yeti/

Müsterinin gereksinimlerini tatmin etmek amacıyla yapay zekay kullanimina hakimiyet için gerekli iki yeti:

- Müsteri gereksinimlerini kavramak
- Yapay zeka destegi olmasaydi bile bunları uçtan uca tek başına implemente edebilip, canlıya alabilecek teknik yeterlilige sahip olmak.

Ilki ne yapılması gerektiğini ve yapay zekanın nasıl yönlendirilmesi gerektiği hususuna işaret ederken, ikincisi hayal ürünü olan ve hicten ortaya cikmis bir ürünü hayat vererek, önü müsterinin hizmetine sunabilmektir.

Yapay zeka ile çalışmak deterministik bir fonksiyon kullanımı gibidir. Yukarda belirttigim iki sabit ve degismez parametreyi bu fonksiyona her verisinizde, beklediginiz ciktiyi alirsiniz, yanı bu iki deterministik parametre, deterministik sonuç üretir. Şimdi neden vibe coding ile hiçbir yere varilamayacaginin teknik sebebini biliyorsunuz.

AI Driven Development

Kaynak: https://kurumsaljava.com/2026/03/18/ai-driven-development/

Yapay zeka araçlarını tam anlamıyla verimli kullandiginiz nasıl anlarsiniz?

Flow olustugunda…

Ben örneğin bir kanatta ajanlarla bir flutter ve web uygulamasını gelistirirken, diğer kanatta bunların ihtiyaç duygugu apileri. backend için başka ajanlarla oluşturuyorum. Bu iki dünya arasında gidis, gelisler bir flow hissi olustuyor yanı doğal bir akis olmaya başlıyor. Siz sadece yazılımcı olarak gereksinimlere odaklaniyorsunuz ve bu perspektiften ajanlara yön vermeye basliyorsunuz. Onlarda size itaat ederek, ihtiyaç duydugunuz yapıları dakikalar içinde olusturuyorlar.

İşte bu akiskan ilerleme hissi olusuyorsa, siz artık yapay zeka güdümlü (AIDD – AI Driven Development) yazılım yapıyorsunuz demektir.

Yapay Zeka ve Context Türleri

Kaynak: https://kurumsaljava.com/2026/03/20/yapay-zeka-ve-context-turleri/

Yapay zeka araçları ile çalışırken üc türlü context var:

- Yapay zekanın kullandigi ve promptlar ile oluşturulan context
- Yazılımcının birden fazla ajani paralelde aynı iş üzerinde calisabilmeleri için kendi zihninde oluşturmak zorunda olduğu zihin contexti.
- Beynin (lislemcinin) paralel calisabimek için kulllandigi context ve onun içindeki context switch

Bahsettiğim ikinci context sadece paralelde birden fazla ajanla çalışmak zorunda kaldigimda ne kadar SINIRLI olduğunu gördüğüm bir yapı. Insan zihni multitasking için yapilmamis. Birden fazla feature üzerinde paralel olaral değişik ajanları koordine etmeye calistigimda, ajanlar arasi gidip gelmeler olduğu için bu zihin contexti islemcideki (beynim) context switchlerden dolayı zor idare edilir hale geliyor.

Bu sebeple kullanmayı ögrendigim en sağlıklı yöntem önce prompt aracılığı ile client tarafinin implemente edilmesini sağlamak ve client ihtiyaclari doğrultusunda backend tarafindaki degisikllikleri yapmak. Bu şekilde zihin içindeki context akisi hep tek yönlü oluyor ve karisiklik olusmuyor. Ama birbirinden farklı iki feature üzerinde caliyorsam, bir noktadan sonra context switch nedeni ile gidisati sağlıklı bir sekide ayristirmam çok zorlasiyor.

Benim cikardigim sonuç şu şekilde:

- Yapay zeka destekli calisildiginda bir feature için genel bir plan olusturulmali.
- Bu plana sadik kalarak küçük kullanıcı hikayeleri (user story) olusturulmali.
- Her kullanıcı hikaye mümkün olan semantik (teknik değil) parçalara bölünmeli.
- Eğer client tarafi varsa, oradan implementasyona baslanmali.
- Client gereksinimleri doğrultusunda backend implemente edilmeli.
- Backend kodu için yapay zekadan entegrarasyon testleri yazmasi istenmeli.
- Ortaya çıkan özelligi test etmek için onay kabul kriterlerini ihtiva eden (user story ye eklenmis olmaları gerekir) onay kabul (acceptance test) testi yazilmali. Bu testi kesinlikle yapay zeka yazmamali. Burada bir test spec yazilarak, test kodun yapay zeka tarafından olusturulmasi saglanabilr.

- Aynı anda sadece bir özellik üzerinde calisilmali.

Karmaşık bir yazılım ürününü “aksam PRD verdim, sabah kalktigimda ürün hazirdi” vari implemente etmeniz imkansız. Profesyonel yaklasim yüksek derecede organizasyon ve interaksiyon gerektiriyor. Bu işi otomatize etmeniz imkansız, çünkü ifade ettiginiz gereksinimlerin yapay zeka tarafından tam anlamıyla anlasildigindan emin olmaniz imkansız. Buradaki ana sorun yine yazılımcının hayal ve ifade kapasitesi.

Yeni Dönemin Programcıları

Kaynak: https://kurumsaljava.com/2026/04/05/yeni-donemin-programcilari/

Yazılımda esas olan kurgu ve akistir. Genel hatlari ile verileri bagimliliklari ile birlikte kurgulayamazsaniz ve birbirleri ile olan iliskilerde veri akisini düzenleyemezseniz, uygulama ya istenileni yapamaz ya da sig bir şekilde yapabilir.

Bu temel prensipleri yapay zeka araçlarını kullaniyor olsaniz bile degistiremezsiniz. Kısaca neyin nasıl yapılması gerektiği hakkında detaylı bilginiz ya da fikriniz yoksa, sizi yapay zeka araçları bile kurtaramaz.

Buradan cikarilmasi gereken başka bir sonuç da, yapay zeka araçlarını bilgisayar mühendisleri, yazılım mühendisleri ve yazılım konusunda tecrübeli insanlar haricinde kimsenin yazılım ile ilgi konularda verimli kullanamayacagidir.

Bu sebeple yazılım bitmemistir sadece başka bir seviye evrilmistir. O seviyenin calisanlari da eski düzende bu ise hakim olan yazilimcilardir. Diğer tayfa zaten işin zorlugunu gördükce ve bu hype sona ermeye basladiginda çok başka islerle (bir sonraki hype) ugrasiyor olacaklar ve bu yazılım olmayacak.

Test Edenler Kazandı

Kaynak: https://kurumsaljava.com/2026/04/11/test-edenler-kazandi/

Ben sahsen “ben programciyim, test yazmam” diyen çok programcı gördüm. Bunlardan hala cokca var.

Şimdilerde ise bu programcıların yapay zekanın ürettigi kodu test etmekten başka ellerinde hiçbir secenek kalmamis olması ne kadar ironik değil mi :)

Test bilgisi, yetisi ve isteklilgi yazılımda her daim iyi yazılımcıyı kötüsünden ayiran bir faktör olmuştur. Yapay zekanın yazilimdaki hakimiye ile test konseptlarine hakim oanlar yine birkaç adim öne gecmislerdir.

Takım Olayı Bitmiştir

Kaynak: https://kurumsaljava.com/2026/04/11/takim-olayi-bitmistir/

Bilyorum bunlar çok radikal söylemler, ama ben daha önceki yazılımcı tecrübelerimi güncel tecrübelerimle kiyasladigim zaman ortaya ckan görüsler bunlar. Gülüp gecebilisiniz. Nasıl olsa kısa bir zaman sonra kendiniz deneyimleyeceksiniz. Ben size hizlandirilmis halini anlatiyorum, yanı spoiler :)

Şimdi şöyle bir sürecten geciyorum. Eskiden promptlarimin çok detaylı, düsük cümlesiz, net ve açık olmasina dikkat ederdim. Zaman içinde bu dogrudan cümlesel bile olmak zorunda olmayan komutlara ya da kelime gruplarina dönüsmeye başladı. Ben çerçeveyi yanı contexti ne kadar net ifade edersem, o oranda yapay zeka benim isteklerimi yerine getirebiliyor, yanı bunun için komple cümlelere bile gerek yok. Benim ifadesel netligim ne kadar azalmis olsa bile, context için gerekli bilgiyi sagladigim sürece yapay zeka benim çok iyi anlayip, işini yapabiliyor.

Şimdi bunu eski proje hayatimdaki çalışma arkadaslarimla olan iliskilerle kiyasliyorum da... Yahu bazen derdimizi anlatmak ya da anlamak için göbegimiz catliyordu. Bu anlasma süreci yer yer çok zaman alici bir hal alabiliyordu. Şimdi ise birkaç kelime bile yeterli ki altında çok karmaşık bir problem olsa bile yapay zeka ne demek istediğimi anliyor ve işini hallediyor.

Sanirim yazılımda takım iside yavaş yavaş son buluyor. Artık tek bir yazılımcı kendi ajanları ile bir takım oluşturacak, gerekmedigi sürece başka yazilimcilarla iletişim bile kurmayacak, çünkü buna zaten gerek yok. Tek işi müsteriyi anlamak ve onun gereksinlerini tatmin eden bir ürün ortaya cikarmaz.

Diyeceksiniz ki entegrasyonlarda mutlaka karşı taraf ile konusmak gerekiyor. Hayır gerekmiyor. Entegrasyonu yazılımcının yapması gerekmiyor. Verin ajana kodu, o gerekli baglantiyi zaten kurar, eğer kuramiyorsa Jira'da bir ticket acip, "şunu şunu şöyle yapin, yoksa kullanilamiyorum abi" der.

Başka ne kaldi geriye iletisimi gerektiren? Ha şu yönetici tayfası var, her ise burnunu sokmaya meraklı. Onlara zaten hiçbir zaman gerek yoktu. Ilk onlar shutlanacak.

Bitti!

Konuyu Bilmiyorlar

Kaynak: https://kurumsaljava.com/2026/04/11/konuyu-bilmiyorlar/

Yapay zeka ile gelinen son nokta...

- En radikal yazılımcılar yazilimi tam anlamıyla birakti ve sadece gereksinimlere ve testlere odaklaniyolar.
- İş yerinde yapay zeka kullanmasi yasak olanlar gizli gizli kendi bilgisayarlarinda kodu yazdirip, google drive ya da başka bir storage üzerinden bu kodu alıp, kopyala, yapistir yapıyorlar. Bunlar aslında birinci grupta olan yazılımcılar.
- Yapay zekayı kesfedenler. Onlar ya kısa zamanda 1. ya da 2. gruba terfi edecekler ya da bir ise yaramadigini düsünüp, vazgececekler. Vazgecmelerinin tek sebebi yazılım konusunda temelde yeterli olmamalari.
- Isittileri kadar konuya vakif olan yazillimcilar. Bunların icinden 3. gruba dahil olma potansiyeli olanlar meraklı olanlari. Meraklı olmayanlar bu işi bırakmak zorunda kalacak olanlar.
- Hiçbir seyden haberi olmayanlar. Onlara Cem Yilmaz'in bu skecini birakiyorum

Hangisi Daha Verimli

Kaynak: https://kurumsaljava.com/2026/04/11/hangisi-daha-verimli/

Hangisi daha verimli?

- Frontend programcısı api gereksinimlerini backend programcisina delege ediyor ve IO veri yapıları üzerinde uzlasma saglamalari gerekiyor. Test, entegrasyon vs derken aradan uzun bir zaman geçiyor ve api tam istenildigi şekilde calismayabiliyor. Bu beraberinde uzlasma seanslari getiriyor. Frontend programcısı backend ve api development konularında kendisini tamamen saliyor ve UI harici bir iş yapmıyor. Aynı şekilde backend programcısı asıl gereksnimlerden bihaber api programlamaya devam ediyor ve gereksinimleri anlama konusunda frontend yazilimcisina bagimli hale geliyor.
- Tek bir programcı hem frontend hem de backend için gerekli çalışmayı yapiyor ve ihtiyaç duyulan apiyi client driven backend tarafında olustuyor. Uzlasmak zorunda kaldigi birisi olmadığı için client için gerekli tüm apileri ihtiyaç doğrultusunda olusturuyor.

Eskiden fullstack denen bir yazılımcı tipi vardı. Full denmesine ragmen ekseriyetle frontend kismindan sorumluydu. Gereksinimleri anlama konusunda backend yazilimcisindan bir adim öndeydi, çünkü müsteriye ya da son kullaniciya daha yakindi. Lakin backend konsunda çok zayifti.

Fullstack rolü tamamen tarihe karismistir. Bunun yerine bir backend yazilimcisi her iki tarafida tek başına koordine edebilecek şekilde 2 numarali satirda bahsettiğim şekilde şekilde fullstack calisabilir.

Çevik Yazılımın Rönesansı

Kaynak: https://kurumsaljava.com/2026/04/11/cevik-yazilimin-ronesansi/

Yazılım dünyasinda taş üzerinde taş kalmiyor :) Doğru bilinenler ve uygulamalar artık kendiliginden ortadan kalkmaya yüz tutuyor. Bunların basinda garip toplantilar var. Digerlerine diğer yazilarimda deginecegim.

Düsünsenize ortada yazılım ekibi bile kalmamis, tek bir yazılımcı kocaman bir ajan ordusunu orkestre ediyor ve toplanmaya gönüllü olanlar sadece yönetici tayfası, çünkü yapacak başka isleri yok. Gözünüzde canlandirin bi... Toplanti odasinda tek bir yazılımcı var ve en az 5-6 tane yönetici, scrum master, menicır, bilmem nerenin baskani vs oturuyor. Sadece toplantilar değil, bu yönetici tayfası da ortadan kalkacak. Ajanlarin onların yerine gecmesi an meselesi :)

Çevik olmak için ne gerek miyordu?

- Yönetici tayfası
- Programciya neyi nasıl yapıldığını söylemek
- Takimi programcı, analizci, testci diye ayirmak.

Çevikliğin özünde ne vardı?

- Yapılacak tüm yazilimla ilgili işi yazılım ekibine devredip, aradan cekilmek
- Yazılımcıların isine karismamak
- Tüm kararlari onların vermesini ve sorumlulugu almalarini sağlamak.

Şimdi bir bakiyorum da, artık yapay zeka ile full çevik yazılım yapabilmek için tüm imkanlar bir araya gelmeye baslamis gibi görünüyor.

Çevik yazılımın rönesansi bu.

Human Context Switch

Kaynak: https://kurumsaljava.com/2026/04/13/huma-context-switch/

Yapay zeka ile çalışırken en limitleyici faktör yazılımcının kendisi.

Yazılımcı birden fazla ajan ile çalışırken baglamlari (context) yönetimesi, context switch nedeni ile önü çok zorluyor.

Örneğin ben bu yazıyı yazarken hem frontend hem de backend ajanları onlara verdigim görevleri yapıyorlar, ama ben kismen hangisinin hangi işi yaptigina dair bilgiyi bu yazıyı yapmak için zihnini toplamak ve context switch yapmak zorunda olduğum için unutuyorum.

Yapay zeka ile çalışırken cözülmesi gereken bir sonraki challenge human context switch. Bu büyük bir ihtimalle yazılımcıyı tamamen devre dışı birakarak mümkün olacak.

Analizin Analizi

Kaynak: https://kurumsaljava.com/2026/04/16/analizin-analizi/

Yapay zeka insanlarda olduğu gibi doğru analiz yapamadigi taktirde karmaşık çözümler üretebiliyor. Şöyle daha kolay olmaz mi dediginizde, haklisiniz çok karmaşık düsünmüsüm, bu benim düsündügüm çözümden çok daha mantikli ve basit diyebiliyor.

Burada öne çıkan durumlar:

- Yapay zaman herşey doğru analiz ve doğru implementasyonu secemeyebilir
- Yazılımcının bu analiz sonuclarini analiz ederek, daha kolay bir çözüm olup, olmadığını kestirebilmesi ve yapay zekayı bu baglamda yönlendirebilmesi gerekebilir
- Yapay zeka tüm contexte hakim olamadigi için ürettigi çözümler çoğu zaman KISS konform olmayacaktir
- KISS olmadan zaman içinde uygulama altyapısı çok karmasiklasir ve bakimi zor hale gelir. Bu yapay zeka ile yeni özellikler ekleme sürelerini drastik bir şekilde artiracaktir.

Sonuç: sadece kod yazmayı yapay zekaya devretmek mantiklidir. Ama o implementasyona giden gerensinim ve çözüm analizi yazılımcının sorumlulugundadir. Ilki vibe coding, ikincisi profesyonel yazilimcilik için gerekli tek yol.

Gereksinimlere Odaklanma

Kaynak: https://kurumsaljava.com/2026/04/16/gereksinimlere-odaklanma/

Eskiden bir uygulama bünyesinde herhangi birşey yapilmak istendiginde süreç yazılımcı için şöyle isliyordu:

Yazılımcı ne yapılması gerektiğini anladiktan sonra, bu gereksinimi hayata gecirmek için teknik detaylar ile boguslaya basliyordu. Bu bogusmaca sonucunda asıl gereksinim yazılımcı odagindan çıkıyor ve teknik çözüm onun yerini aliyordu. Bu sebeple istenilen değil de teknik olarak yapilabilen ortaya cikiyordu çoğu zaman.

Yapay zeka ile bu odak kaynamasi artık gerekliligini yitirdi, çünkü yazılımcının teknik detaylar ile ugrasma zorunlulugu azaldi ve bu mental yükün hafiflemesi ile tamamen yazılımcı müsteri gereksinimlerine odaklanabilmekte.

Yazılımda esas olan zaten budur, müsteri gereksinimlerine odaklanma!

Dağarcığın Gücü

Kaynak: https://kurumsaljava.com/2026/04/17/dagarcigin-gucu/

Yapay zaka ile ortaya çıkan işin kalitesi, yazılımcının yetenekleri ile dogrudan iliskilidir. Yazılımcı neyi yapabiliyorsa yanı neye kadir ise, önü yapay zekaya yaptirabilir. Buradaki tek fark; kendisinin yapmıyor, yaptiriyor olmasıdır.

Yazılımcı kendisinin tek başına olusturamayacagi bir çözümü yapay zekaya yaptiramaz. Yaptirsa bile bu tesadüfi çözümü ne yönlendirebilir ne de anlayabilir.

Kelime hazinesinin genisliginin ifade gücü üzerindeki etkisi ne ise, yazılımcının yetenek, bilgi ve tecrüsinin yapay zeka destekli çözümlerin basarisi üzerindeki etkisi de odur.

Gerçek hayatta olduğu gibi yapay zekada da önü kullananin dagarciginin gücü sonucu belirler.

Zayıf Bağlam

Kaynak: https://kurumsaljava.com/2026/04/17/zayif-baglam/

Yapay zeka bağlam sorunu nedeni ile alzheimer hastasi gibi yaptığı çoğu şeyi unutabilir.

Bu küçük çaplı islerde sorun teskil etmiyor. Lakin bir sorunu çözmek için daire içinde dönmeye basladiginda, zayıf bir bağlam yapay zekanın birbiri arkasina hep aynı çözümleri uygulayarak, değişik sonuclar almaya calismasina sebep olabilir.

Burada yazılımcının bu döngüyü kirmasi ve yapay zekaya daha önce kullandigi çözümlerin hatirlatilmasi, bunların bir ise yaramadiginin, sorunu tekrar başka bir perspektiften degerlendirmesi gerektiği söylenebilir. Ayrıca “aynı hatalar olusuyor, dairede dönüyoruz, sorun neden çözülmüyor” gibi sözler de yapay zekayı sorunu değişik acilardan incelemeye zorlayabilir.

Bundan sonra masasinda ajanlarina bagiran, cagiran, onlari zorbalayan yazılımcı görürseniz, sasirmayin :-)

Lütfen

Kaynak: https://kurumsaljava.com/2026/04/17/lutfen/

Opus-4.7 daha bi özgüveni yüksek mi ne?

Bir sorunu analiz etmesini istedim. “Tam stack lazım, yoksa sallamis olurum” diyor :).

Opus 4.6 olsa “stack lazım, verebilir misin” diye sorardi.

Yakinda o sihirli sözü söylemedin ama filan der bu.

Biraz önce ilk defa 3 gün süren bir rate limit yedim. Hemen başka bir hesap acip devam ettim. Opus’un kafasi bozulunca rate limit ile tehtit eder giibime geliyor ;-) O yüzden artık her prompt lütfen ile başlayacak.

Ben tirstim iyice. 3 bün ban ne demek ya! Bütün işin alt-üst olur. Günün sonunda kendim yaparım da, bu rahatliga alisinca artık herşey daha zor. Hiçbir şey eskisi gibi olamaz artık.

Ne günlere kaldik (geldik) ya!

Claude Opus

Kaynak: https://kurumsaljava.com/2026/04/18/claude-opus/

Opus 4.7 ile deneyimlediklerim:

Yeni Opus'dan 4.6 ile yaptirdigim karmaşık bir veri senkronizyon implementasyonunu analiz etmesini ve aciklarini bulmasini istedim. Yirmiye yakın madde iceren bir liste olusturdu. Bunlardan dört tanesi çok kritik. Özellilkle bunlardan ilki bir güvenlik sorunu.

Bu hatalarin critical ve high olanlarini ortadan kaldirmasini istedim. Sekiz saat süren bir çalışma sonunda mevcut implementasyonu da çalışmaz hale getiren bir sonuç ortaya cikardi, çalışan seylerde çalışmaz oldu.

Bu noktada Opus 4.6 birkaç iterasyon ile yaptığı ya da buldugu hatalari düzeltebiliyorken, Opus 4.7 halüsinasyon sebepli gereksiz, hatali ve anlamsız analizler sonunda implementasyonu daha da kötülestirdi.

Opus 4.7 çok iyi analiz yapabiliyor, lakin önerilerini karmaşık mimari ve yapidaki bir kod tabaninda implementasyon ile taclandirmakta çok zorlaniyor. Burada islemci gereksinimini minimize edebilmek için context kapasitesi üzerinde degisiklige gidilmis gibi duruyor. Bu yüzden Opus 4.7 benim güvenimi yitirdi.

Bundan böyle ben implementasyonu Opus 4.6 ile yapmaya devam edeceğim. Akabinde Opus 4.7 ile mevcut aciklari analiz ettirip, bunların yine Opus 4.6 tarafından islenmesini saglayacagim.

Opus 4.7 Opus Mythos'un temelini olusturuyor büyük bir ihtimalle, çünkü critik güvenlik sorunlarini bulmakta çok yetkin.

Amiga Efekti

Kaynak: https://kurumsaljava.com/2026/04/18/amiga-efekti/

Amiga 500 sahibi olanlar bilirler. Ram extension slotu vardı. 512KB olan hafizayi yine 512KB bir kart ile 1MB yapmıştım.

Yapay zeka ile böyle bir analoji yasiyorum. Şöyle ki:

Temelde yapay zeka yazılımcının contextini büyütmüs oldu.

Yazılımcı bu yeni context ile daha karmaşık yapılara hükmedebiliyor, daha fazlasini görüyor, analiz edebiliyor ve daha hızlı karar verebiliyor.

Aslında yapay zeka yazilmcinin mevcut kapasini artirmak için kullanabilecegi bir ram ve cpu extension karti.

Ben Amiga bilgisayarimin hafizasini 1MB ye yükselttigimde hiçbir değişiklik hissetmemistim. Ama yapay zekada durum çok çok farklı ;-)

İşte yapay zeka bazilarinda Amiga efekti olustururken, bazilarinda level atlatan bir basamaga dönüsecek.

Nyet!

Kaynak: `https://kurumsaljava.com/2026/04/18/nyet/`

Tribünden bir futbol maci seyrederek, iyi bir futbolcu olunabilir mi? Nyet!

Teknolojik gelismeleri takip ederek, iyi bir yazılımcı olunabilir mi? Nyet!

İnsanın yaptiginda iyiye götüren tek birşey var: Just do it!

Teknolojik gelismeler büyük bir dalga gibi gelirler. Surf yapanlar o dalganın gücünü kullanirlar. Dalganın karsisina gecip, izlerseniz, gelir ve bir zaman sonra cekilir. Dalganın üzerindeyseniz, sizi tasir. Aynı şey yazılım teknolojileri için de gecerlidir.

Ögrenmek ve yapmak iki kardes gibidir, yapmak ögretir, ögrenmek için yapmak gerekir.

Yazılımda Döngüler

Kaynak: https://kurumsaljava.com/2026/04/18/yazilimda-donguler/

Yapay zeka ile yazılım yaparken benim döngülerim

- Gereksinimleri küçük parçalara bölüp, gerekli promptlar ile yazilimi yaptırıyorum.
- Yapay zeka entegrasyon testleri yazarken, ben onay/kabul kriterleirni ihtiva eden test specleri hazirliyorum. Bu specleri test koduna yapay zeka çevirip, koşturuyor.
- Implementasyon esnasinda kodu kabaca gözden geciriyorum ve yapay zekanın her prompt sonrasında sundugu çözümü gözden geciriyorum.
- Implementasyon düsündügüm şekilde calisiyorsa yanı uygulama blackbox olarak beklentilerimi karsiliyorsa, implementasyonu tamamlanmış sayiyorum.
- Daha güclü bir model ile otomatik code review yaptirip, açık ya da buglari analiz ettiriyorum.
- Bulunan buglar için tekrar gerekli promptlar ile kodsal düzeltmelere gidiyorum.
- Çıkan kodu tekrar daha güclü olan model tarafından analiz ettiriyorum.

Bu islemlerin sonunda tüm onay/kabul testleri, entegrasyon testleri ve manuel testler calisiyorsa, kod prod ready statüsüne geçiyor ve deploy ediliyor.

Bitti mi?

Kaynak: https://kurumsaljava.com/2026/04/19/kontext-kapsama/

Yapay zeka bir uygulama özelligi tamamladiginda, bilin ki o özellilk daha çok eksik.

Yapay zeka sahip olduğu context kadar duruma hakim olabilir. Bilgi ve veri eksikliginden dolayı durumu 360 derece analiz etme ihtimali çok düsük. Bu yüzden bir uygulama özelliginin kapsamini yaziimci tayin etmek ve akabinde kontrol etmek zorundadır.

Siz bitti dediginizde bile bitmemis olabilir. Burada yapay zekanın mevcut durumu analiz etmesini istediginiz taktirde, eksik kalmış yanlari size göstebilir. Buradaki en önemli ana unsur contextin kapsayicilik özelligidir. Context içinde ne kadar çok net bilgi varsa, oluşturulan uygulama özelliginin o oradan tamamlanmış olma itimali yükselir.

Burada benim cikardigim sonuç şu: yapay zeka ile iteratif çalışmak gerekir. Her yeni bir iterasyon ile her iki taraf için de yapılması gerekenler daha da netlesecektir. Her yeni iterasyon contexxt kapsama oranini artırır.

Yapay Zekaya Güvenmek

Kaynak: https://kurumsaljava.com/2026/04/19/yapaya-zekaya-guvenmek/

Yapay zeka ile uygulamayı cikardiniz ama kodun durumu hiç icinize sinmedi mi?

Akliniza birçok soru geliyor olabilir. Örneğin yük altında stabil calisak mi? Paralel islemler veriler üzerinde hata izleri birakacak mi vs. Eğer akliniza hiçbir soru gelmiyor ve uygulamayı hemen canlıya almak istiyorsaniz, bu yazıyı önce okuyun derim :)

Yazılım sadece kodun çalışıyor olması ile bitmez. Localhost üzerinden tek bir kullanıcı ile test edilmis bir uygulama henüz hiçbir seye hazir değildir. “Lokalde çalışıyor ama” söylemlerinin ana sebebi de budur.

Yapay zekanın ürettigi kodu nasıl güvenilir hale getirirsiniz? Burada benim gibi artık hiç kod yazmadiginizi farz ediyorum. En alttan yukariya doğru atilmasi gereken adimlari siralayacagim.

** Birim Testleri Her prompt içinde mutlaka “test etmeyi de unutma” ibaresi olmali. Yapay zeka oluşturduğu her satır kod için karsiligi olan bir birim testi yazacaktir.

** Entegrasyon Testleri Bu testler entegre edimis bir sistem de (api + db) mevcut apilerin onay/kabul kriterlerine uygunlugunu test ederler. Bu tür testlerde yapay zeka tarafından olusturulmalidir.

** Code Coverage Mevcut testlerin ne oranda kodu kapsadigini anlayabilmek için code coverage araçları ile ölcüm yapılması gerekmektedir. Ben yapay zekaya arada bir “kod kapsama seviyemiz” nedir diye soruyorum. Aldığım cevap genelde tüm sistem geneli için geçerli olan %90 civarinda oluyor. Ayrıca yapay zekadan mevcut testleri inceleyerek, hangi alanlar için test yazilabilecegi analizini istiyorum. Bu şekilde zaten eksik olan tüm birim testleri ortaya cikmis oluyor.

** Onay/Kabul Testleri Yazılımcı bu noktadan itibaren uygulamayı test etme islemini kendi eline almak durumundadir, çünkü uygulamadan olan beklentilerin onay/kabul kriterleri ile net olarak ifade edilmeleri gerekmektedir. Burada yine given/when/then tarzı onay/kabul kriterlerinin yer aldigi tanimlamalar ihtiva eden test spec dosyaları olusturulur. Bu spec dosyalarinin test koduna dönüstürülmesi gerekmektedir. Burada yine yapay zekadan bu spec dosyalarindan test kodunu oluşturacak minik bir test çatısı oluşturması istenir. Buradaki en büyük avantaj beklentilerin plain text ile ifade edilmesi lakin gerekli test kodunun olusturulma isleminin yapay zekaya birakilmadir. Test kodu dogasi itibari ile çok kirilgan bir yapiya sahiptir. Yazılımcının test kodu yazmak yerine, beklentilerini

ifade ettiği spec dosyaları oluşturması, kirilgan olan test kodun bakim yükümlülügünü yapay zekaya aktarmaktadir.

** Test Harness Uygulamanın karmaşık bir kısmı olduğunu düsünelim. Ben burada kendi calismamdan bir örnek vereyim. Cloud main backend ile müsteri isletmelerinde benzer yapıda çalışan custom onprem backend sistemlerinin karsilikli olarak verisel senkronize edilmesi gerekiyor. Onprem tarafından örneğin oluşan siparislerin düzenlli olarak cloud main backende aktarilmasi lazım. Aynı şekilde cloud main backend üzerinde yapılan tüm değişiklikler verinin ait olduğu onprem instance tespit edilerek SSE üzerinden oraya aktarilmasi gerekiyor. Burada çok karmaşık bir replikasyon implementasyonu var.

Bu yapiyi onbinlerce sipariş ve benzeri verilerle test edebilmek için yapay zekadan bir test harness çatısı oluşturmasını istedim. Bu test harness bünyesinde onbinlerce siparisi otomatik olarak onprem üzerinde olusturuluyor ve akabinde bu verilerin cloud main backende intikal edip, etmedikleri test harness çatısı tarafından kontrol ediliyor. Bu bir yük testi değil, ben sadece onprem veritabanında oluşan her satirin doğru bir şekilde cloud main backend aktarilmasini test etmek istedim.

Bu şekilde karmaşık uygulama özellikleriin yük altında ne kadar doğru ve stabil calistiklari test edilebilir.

Eğer çıkan sonuclara güvenmiyorsaniz, benim yaptigim gibi yapay zekadan iki veritabanini sync log dosyası isiginda kiyaslamasini isteyebilirsiinz. Gün sonunda hakikaet veritabanindaki veride yatmaktadir.

** Yük Testleri

Load ya da performance testing için birçok araç mevcut. Bunun için yapay zeka destegi almak gerekmiyor diyebilirsiinz, ama çıkan sonuclarin analizi için yine yapay zeka kullanilabilir.

** Uygulama Loglari

Tek bir log yerine her büyük islem için bir log dosyası tahsis edilebilr. Benim uygulamada örneğin sync işlemleri, sipariş transaksiyonlari, merge işlemleri, sse watchdog için ayrı birer log dosyası var. Buna paralel tüm loglarin toplandigi ana bir log dosyası da mevcut, lakin bir hata bulmak için bu dosyayi yapay zekaya verip, contexti patlatmanin ve tokenleri yakmanin bir anlami yok.

Bunun yerine hem hata aramak hem de implementasyonu check etmek için küçük log dosyalarını yapay zeka ile paylaşıyorum. Bu şekilde çok implementasyon hatasi kesfedebildim.

** Başka Modelleri Ile Cross-Check Ben implementasyon için opus-4.6, analiz için

opus-4.7 kulllaniyorum. 4.7 kod yazma konusunda biraz ucuk, ama analizleri çok iyi. Bu yüzden analizleri yüksek modelle yapip, implementasyon için bir alt modeli kullanıyorum. Bu şekilde opus-4.6 nin aklina bile gelmeyenleri, eksikleri ya da hatalari opus-4.7 ile kesfetmek ve tamamlatmak münkün. Bu yüzden “güven iyidir, ama kontrol daha iyidir” demek geliyor icimden :)

** Sürekli Entegrasyon Her commit ile kodun derlenmesi ve mevcut testlerin kosturulmasi gerekir. Eğer test kirilmalari varsa, bu yapısal bir degisikligin yan etkilerinin olduguna işaret eder. Bu yüzden sürekli entegrasyon (continuous integration) kullanilmasi gereken önemli süreclerden birisidir. Tüm testlerin çalışıyor olması yeni bir sürüm olusturmanin ilk adimi olmalidir.

Frontend First

Kaynak: https://kurumsaljava.com/2026/05/05/frontend-first/

Yapay zeka destekli çalışırken uçtan uca bir uygulamayı gelistiriyor olmak çok önemli. Kesinlikle frontend için ayrı, backend için ayrı analiz ve implementasyon yapilmamasi gerekiyor. Zaten klasik çalışma yöntemlerinde ekipleri en çok meşgul eden durum da bu olmuştur; kendi baslarina, koordine etmeden iş yapmalari ya da backendin arayüzleri istediği şekilde yönlendirmeye calismasi.

Örneğin backend tarafında gerekli olduğunu düsündügünüz yeni bir özelligi implemente etmeye baslamayin, çünkü arayüzün ihtiyaclarini tam olarak bilemediginiz için bu analiz / implementasyon eksik olacaktır. Bu yüzden öncelikli olarak mock vari bile olsa önce client konumunda olan arayüzün (frontend) tasarlanmasi ve ihtiyaclarinin öncelikli olarak implemente edilmesi gerekmektedir. Arkasi zaten corap sökügü gibi gelecektir.

Kendi projemdem küçük bir örnek vereyim. Tamamlanmış bir sale için refund yapılması gerekiyor. Bunun için şu anki durumumuzu kiyaslayarak nelerin eksik olduğunu ve nasıl implemente edilmesi gerektiğini ögrenmek amacli bir analiz baslattim. Gelen çözüm önerisi makul duruyor, lakin gerçekten bu kadar derin degisiklige ihtiyacımız olup, olmadigina dair aklimda soru isaretleri olustu. Eğer devam etseydim arayüzü bu değişiklikleri kullanmaya mecbur birakacaktim. Bunun yerine diğer yolu yanı topdown (yukaridan aşağıya) yönetimi uyguladim. Arayüzü olması gerektiği şekilde tasarladim ve gerekli apileri ve api degisikliklerinin backend kanadinda implemente edilmesini sagladim. Bu şekilde sadece backend tarafında gerçekten ihtiyaç duyulan yapılar olusmaya başladı ve ilk basta sunulan analizin overengineered olduğunu gördüm.

Yazılım geliştirme sürecinde takip edilmesi gereken bir patika var. Bu patikanin basinda müsterinin kendisi yer aliyor. Müsterinin gereksinimleri doğrultusunda arayüzler olusturulur. Akabinde bu arayüzlerin ihtiyaç duydugu backend apiler implemente edilir. Backend kesinlikle bir servis katmanidir ve frontend ne istiyorsa, sağlamak zorundadır. Bakmayin siz backend yazılımcıların bu kadar havali olduklarina :) Evet zor kisim backend ama patron her daim frontenddir. Frontend ne isterse, o yapilir. Gereksinimlere hakim olanlar da frontend yazilimcilaridir.

Yapay zeka ile bu ayrim artık ortadan kaltigina göre, “frontend first” ile baslanmasi müsterinin gereksinimlerini tatmin edecek en iyi cözüme götürecektir.

Yapay Zeka İle Kaybolacak Meslekler

Kaynak: https://kurumsaljava.com/2026/05/10/yapay-zeka-ile-kaybolacak-meslekler/

Bilindigi üzere yapay zeka aniden hayatimiza girdi ve birçok şeyi köklü bir şekilde degistirdi. Bu değişim e fonksiyonu gibi çok daha hızlı bir ivme ile devam edecek.

Bunu büyük bir firsat olarak algilayanlar yaninda, büyük sikintilarin bizi bekledigini düsünenler de var. Birçok meslegin yok olacagi ve insanlarin issiz kalacagi söyleniyor. Ben size kimlerin issiz kalacagini ve sadece bunun insanoglunu yasamsal anlamda ne kadar büyük bir darbogaza sokacagini söyleyeyim mi?

- Avukatlar
- Doktorlar
- Pilotlar
- Muhesebeciler

değil!

Onlar yokken de insanlar yasamlarini sürdürüyorlardi.

Asıl kaybolacaklarlar ya da yetismeyecekler:

- Sairler
- Filozoflar
- Yazarlar
- Düsünürler

Şimdi şöyle bir elli senesi sonrasina gidin, kimsenin edebi ve filozofik eserler ortaya koyamadigi (çünkü hiç kimsenin artık bu konularla ilgilenmedigi ve yapay zekanın onlara verdigi yüzeysel özetlerle yetindikleri için) bir dönemde insanlara gelecege yönelik neyin ışık tutacagini düsünün. Ne görüyorsunuz?

Sizde insanoglu bu şekilde daha mi hızlı gelisecek yoksa daha hızlı mi taş devrine geri dönecek?

İnsanın biyolojik evrimini bir kenara birakirsak, gerçek anlamda zihinsel sicramasi bahsettiğim edebi ve filozofik eserler üzerinden olmuştur. Günümüzde bile bu eserlere talep çok az iken, yapay zekanın gelismesi ile bizi çok büyük tehlikenin bekliyor olduğunu görmek artık zor değil.

Sair, filozof ve yazar yetistirmeyen toplumlar olaylari sorgulama yetileri olmadığı için ya yok olurlar ya da kölelestirilirler.

Opus 4.8 ile İlk Deneyimlerim

Kaynak: https://kurumsaljava.com/2026/06/07/opus-4-8-ile-ilk-deneyimlerim/

Opus 4.8 ile çalışmak bir tuval üzerinde firca sallayan bir ressam gibi hissettiriyor. Bu his en son Opus 4.6 ile çalışırken olumustu ve Opus 4.7 ile tamamen ortadan kaybolmustu.

Opus 4.7 ile teknik detaylara çok odaklanmak zorunda kalirken, Opus 4.8 ile yeniden tamamen, asıl isim olan gereksinim analizi ve onun gerceklesmesine odaklanabiliyorum. Opus 4.8 ile bir şeyi ikinci kez tekrar etmek ya da tamamlayici nitelikte tekrar tanımlamak zorunda kalmadım. Implementasyonlari nokta atisi yapabiliyor.

Opus 4.7 ile çalışma performansim ve günlük islem kapasitem düsmüstü, çünkü istediğim şekilde ilerleyemiyordum. Opus 4.8 ile yine o eski akisi (flow) yakaladim ve saatlerin nasıl gectigini farketmiyorum.

Opus 4.8, Opus 4.6 implementasyon yeteneklerinin ve Opus 4.7 analiz kabiliyetlerinin birlesmis hali.

Not: Eskiden Github Copilot CLI kullaniyordum. Github'un (Microsoft kaynakli) ucuklasan fiyat ve kullanim politikaları sonunda, yaka silkerek Claude Code ve max20 ye geçtim. Verdigim en iyi kararmis.

AI First

Kaynak: https://kurumsaljava.com/2026/06/08/ai-first/

Çok karmaşık bir yapı üzerinde calisiyorsunuz. Yapay zeka (benim örnegimde opus 4.8) gerekli implementasyonu gerceklestirdi ve sonuç istediginiz şekilde olmadi. Ne yaparsiniz?

Örnek olması acisindan benim bu durumda nasıl ilerledigimiz aktarayim.

Kesinlikle debug yapmaya baslamiyorum. Benim zihnim artık tamamen code first değil AI first olarak isliyor. Böyle durumlarda yeni bir prompt ile yapay zekaya durumu arastirmasini söylüyorum. Eğer bu arastirma sadece kod bazinda olursa, yapay zeka hatayı bulamayabilir. Zaten bulabilme potansiyeli olsaydi, implementasyonu doğru yapardi.

Burada, oluşan data set ile çalışmak daha mantikli. Yapılan islem sonunda veritabanında birtakim recordlar olustu. Prompt yazarken hatayı net olarak tanımlıyorum, akabinde "veritabanında örneğin şu ismi tasiyan bir ürün var, ama parçaları eksik, verilerden yola cikarak neden eksik olduğunu arastir ve hatayı cöz" diyorum.

AI first ile kastim bu; tüm kod yazma, hata bulma, debugging gibi islemlerin tamamen yapay zekaya devredilmesi.

Neden bu şekilde çalışıyorum? Teknik taraf için gerekli mental yükün altina girmemek için. Ben zaten yapay zekanın yaptiklarindan eminim. Benim görevim sadece ortaya çıkan neticeyi yanı uygulamanın sahip olması gereken davranış bicimlerini check etmek. Hata buldugumda debug yapmaya baslarsan, o zaman oluşan tüm yeni yapılara hakim olmak için herseyin üzerinden gecmem gerekir ki bu da bende mental yükü artırır. Zaten debug yapmaya baslamak bile basli başına büyük bir mental yüktür.

Kodun nasıl sekillendigi beni ilgilendirmiyor. Ben gerekli mimari çerçeveyi olusturdum ve yapay zeka zaten bunları örnek olarak yeni yapıları olusturacaktir.

Kisacasi bir ustanin isine karismamak lazım. Usta, bu çalışmıyor denilerek belki yönlendirilir ama kimse evine gelen bir tesisatci ustasinin problemi cözemedigi noktada eline onun cekicini alıp, duvara delik acmaya çalışmıyor, öyle değil mi?

Günlük yasantimizda doğal olan yanı bize doğal gelen sürecleri kendi çalışma tarzimiza da adapte etmemiz gerekir. Eğer yapay zeka ile calisiyorsam, tüm sorumlulugu ona birakmam gerekiyor. Elime cekici alıp, bu olmamis diyerek, araya girmek işin mantığına aykiri.

Bu sebeple ya ai first ya da hiç o topa girmeyeceksiniz.

Yazılım Hala bir Zanaat mı?

Kaynak: https://kurumsaljava.com/2026/06/08/yazilim-hale-bir-zanaat-mi/

Şöyle bir yazım var, okuyanlar bilirler: Programcılık sanat mi, zanaat mi?

> Programcılık Sanat mı, Zanaat mı?

Bu yazım LLMler ünlenmeden ve benim yazılım konusunda fikirlerim degismeden önce kale aldığım bir yaziydi. Ben yazilimi hiçbir zaman sanat olarak görmedim. Yazılım ustalik gerektiren birşey ve bir zanaattır. Bazilarinin idda ettiği gibi siir gibi kod yazinca sair, yanı sanatci, o kod da sanat eseri olmuyor. Sanatin islevi olmaz, ama kodun islevi var ve istenildigi kadar kopyalanabilir (reproduction), ama sanat eserlerinden genelde bir adet ya da çok az sayida vardır.

Şimdi bu konuda fikrim kismen değişmiş durumda. Programcılık hala bir zanaat, lakin bu kisim LLM'e ve içinde bulundugu ekosisteme kaymis durumda. Bizler yazılımcı olarak dirigent konumuna geldik. Ben burada kendimi bir ressam gibi hayal ediyorum ve önündeki bir tuvali (uygulamayı) istediğim firca darbeleri (promt) ile sekillendiriyorum. Bu baglanda yaptigim biraz sanati cagristiriyor. Ama sanat ile bagi sadece bundan ibaret. Bir dirigent olarak yine sistemi bir bütün olarak tasarlayip, çalışır hale getirmem gerekiyor. Bu da yine bahsettiğim o ustaligi ve beraberinde zanaat faaliyetini getiriyor. Yanı yazılım yapmanin dogasi zanaattır.

Full Automated Coding

Kaynak: https://kurumsaljava.com/2026/06/08/full-automated-coding/

Şimdi yine reklam yapmaya başladı denecek, ama yine de söylemek zorundayim:

Opus 4.8'e hemen gecip, bütün yazılım sürecini ona devretmeyen

- Mevzuyu hala anlamamıştır
- Hiçbir AI tool tecrübesi yoktur
- Başka araçlar (codex gibi) ile kötü tecrübesi olmuştur ve AI olayini rafa kaldirmistir, çünkü daha iyisini kendisi yapabiliyordur
- İş yerindeki güvenlik politikaları (ona, biz de ne olduğunu anlayamadik henüz politikaları diyelim) yüzünden AI araçlarını kullanamiyordur.

Eğer sonuncu söz konusu ise, o isten hemen ayrilir, maas, muusa bakmadan yapay zekanın full entegre kullanildigi bir ise gecerdim. Aynı iş yerinde daha yıllarca kalarak, bu nimetlerden tamamen bihaber kalma ihtimali çok yüksek. Bindiginiz dalı kesmek üzeresiniz.

Yazılımın Opus 4.8 ve henüz bizim görmedigimiz el altindaki modeller ile artık belli olan tek bir yöne doğru gidiyor: full automated autonom implementation by ghosts.

Bu terimi ben uydurdum, ama ne demek istediğimi anladiniz. Geriye sadece bu sistemleri yönetecek programcılara ihtiyaç kaliyor. Onlar da masters of the AI ecosystem olacak, yanı çok hızlı bir şekilde o ekosistemlerin kullanımı üzerine ihtisas gerekli.

Eğer bu yolda kendi basima nasıl ilerlerim diyorsaniz, yol haritasi bu:

- Sokaga çıkıp, insanlari meşgul eden bir problem kesfi
- Claude code ve max subscription
- Sorunu cözmek için opus 4.8 ile bir satır kod yazmadan ürünü geliştirme
- Ürünü hemen yayilabilmesi için ücretsiz dagitma
- Ürün etrafinda hizmet sunma ve para kazanma.

Ben bunu yapıyorum şu anda!

Hadi ürün tutmadi diyelim ki %99 tutmayacaktir, ama siz yapay zekanın bir kenarindan tutmus oldunuz ve bir ürünün nasıl gelistirildigine uçtan uca taniklik etmis oldunuz.

Ufak bir not daha düseyim: Burada vide coding mevzusundan bahsetmiyorum. Bunu yapabilecek kisilerin sağlam bir enformatik altyapısı olması gerekiyor. Mühendisligini okumadiysaniz ya da yillardir programcı olarak calismiyorsaniz, zaman kaybindan başka birşey olmazdi.

Yapay Zekaları Birbirlerine Kırdırma

Kaynak: https://kurumsaljava.com/2026/06/09/yapay-zekalari-birbirlerine-kirdirma/

Siz hiç iki yapay zekayı birbirine kirdirdiniz mi? Çok eglenceli.

Kırdırma biraz egzejere bir örnek oldu. Daha ziyade birbirlerinin saglamasini yapma diyelim buna.

Bir örnek üzerinden inceleyelim.

Prompt hazirladim ve bir feature için Opus calismaya başladı. İşi netlestirmesi için AskUserQuestion ismini tasiyan bir araci kullanmasini istiyorum. Bu durumda Opus varsiyamlar yapmak yerine bana sorular sorarak, analizinde ilerliyor.

Sordugu sorularin cevabını biliyorum. Soruları bazen dogrudan cevaplamak yerine, bu soruları ChatGpt'ye yöneltiyorum. Eğer aynı fikirdeyse, sorun yok, ama farklı bir fikir ortaya cikiyorsa, bu sefer ondan aldığım bilgileri Opus'a veriyorum ve onun fikirini alıyorum. Eğer temelde bir fikir ayriligi varsa, birbirerini yiyorlar tabi, ama bu çok ender olan bir durum. Aklin yolu bir olduğu için çok hızlı bir şekilde uzlasiyorlar.

Almanca'da bir deyim var: der klügere gibt immer nach -> akilli olan daha esnek davranir. Acaba bunu yapay zeka düstur edinmis olabilir mi? Çünkü çoğu zaman aynı fikirde olmaları biraz garip, çünkü birbirlerinden haberleri yok, aynı kod baze üzerinde calismiyorlar, sadece fikirsel olarak tartisiyorlar. Bilemeyiz ;-)

İpek Böceği

Kaynak: https://kurumsaljava.com/2026/06/09/ipek-bocegi/

Tüm yazilimi yapay zekaya birakmis birisi olarak, yetilerimin körelmemesi için ne yapıyorum? Hiçbir şey!

Yıllar harcayarak kazandiginiz o yetiler kullanmadiniz diye kaybolmazlar. Bisiklet sürmesini ögrenmis birisi her daim o bisikleti sürecek yetiye sahiptir.

Yapay zekayı verimli kullanan bir yazılımcı (senior) algoritmalari, veri yapilarini, mimarileri zaten bilen, kullanan, o konular hakkında yeni gelismeleri takip eden birisidir. Kendisi artık algoritma yazmiyor olsa bile, algoritma okuyabilir, mimari olusturabilir vs.

Zaten ters mantık şu değil mi? Ben neden artık algoritma yazma yetimi korumaliyim ki? Bunu benden çok daha iyi yapan bir araç var elimde, o yapsın! Ben zaten yaptıklarını okuyabilmek, anlayabilmek, degerlendirebilmek, cöpe atip, yeniden yaptirabilme yetisine sahibim. Neden ben daha iyi algoritma yazma cabasi içinde olmaliyim? Olmamaliyim!

Aynı zamanda hem çok iyi bir programcı hem de çok iyi bir yapay zeka araci kullanicisi olmaniza gerek yok. Iyi programcılık vasfini artık yapay zekaya birakma zamanı geldi. Siz yapay zeka kullanirken daha iyi bir programcı olmaliyim fikrini tasimaya devam ediyorsaniz, kapasitenizin çok büyük bir kismini doğru yerde kullanmiyorsunuz demektir. Kullanmayacaginiz yetilere yatirim yapmak yanlıştır. Bunun yerine bir metamorfik dönüşüm içinde yapay zekaya her alanda hükmetmek için gerekli tüm yetileri kazanma savasi vermek daha mantiklidir. Gerekli yetiler artık iyi kod yazmak, algoritma kurmak, mimari olustmak değildir. Yeni yetiler gereksinim analizi, müsterinin isteklerini anlamak, yapay zekayı yönlendirmektir. Bu konularda uzmanlasmaniz gerekiyor. Digerlerini artık araçlar çok daha iyi yapiyor. Orada gelistirebileceginiz kas kalmadı. Teknik konulara çok calisarak hipertrofinin size birşeyler katmasini beklemeniz anlamsız bir cabadir.

Bundan böyle bir satır kod yazmayacak birisinin iyi bir programcı kalmasi fikri cöptür. Tabi ben burada yeni yetişen programcilardan bahsetmiyorum. Temeli sağlam senior pogramcılar için geçerli söylediklerim. Onların zaten dogalari gereği herşeyi merak ettiklerinden, zevk aldiklari alanlarda (algoritma, mimari vs) kendilerini geliştirmeye devam ederler, çünkü bunu zorunluluk olarak değil, eglence olarak görürler. Benim bahsettiğim konu daha ziyade, “illa A, B , C yetileriinin körelmemesi gerekir, onlar üzerine calis” deyiminin zirva olmasıdır.

Marifet gereklilikleri görüp, önü tatmin edebilecek yeni yapiya dönüsmektir, aksi

taktirde insan olduğu yerde sayar ya da biraz bundan biraz ondan yapar, ama tam anlamıyla calistigi alana hakim olamaz.

Bir ipek böcegini düsünün, ne demek istediğimi daha net anlayacaksiniz. Bir kelebek aynı zamanda bir kelebek, bir ipek böcegi ya da ipek güvesi değildir. Bir kelebek kelebektir, ama gectigi o sürecleri kendi içinde tasir. Kelebek artık ucmaya odaklanir, koza örmeye değil. Yapay zeka ile çalışmak isteyen bir senior programcı da artık iyi bir programcı olma derdinden vazgeçmek zorundadır., çünkü artık programcı değildir.

Yeni Derleyiciler LLM'ler

Kaynak: https://kurumsaljava.com/2026/06/09/yeni-derleyiciler-llmler/

Artık kod yazmaya gerek kalmadı söylemlerime sürekli dolaylı ya da dogrudan itirazlar geliyor. Yanı bir yerlerde mutlaka lazım olacaktır, biz yazmaya hazir olalim minvalinde söylemler genelde.

Hayır, kod yazmaya gerek kalmadı! Nedenini açıklayayım.

Ben ilk kod yazmama deneyimlerime Opus ve Sonnet 4.5 ile başladım. Sonnet bu konuda çok başarılı değildi ve ben çoğu yeri elden tamamlamak zorunda kaliyordum. Daha sonra Opus 4.5 geçtim ve durum da daha da kod yazmamaya doğru gitti. Opus 4.6 ile durum tamamen değişti ve ben neredeyse %95 lere geldim kod yazmama isinde. Akabinde Opus 4.7 hayal kirikligi yaratsa da %90 larin üzerinde kodun yazilma işini ona bıraktım. Şimdi Opus 4.8 ile çalışıyorum ve bu oran %100. Ilk promptu 4-5 gün önce verdim ve o günden sonra 1 satır kod yazmadım ve bu böyle devam ediyor.

Çoğu programcı kendisini kod yazmak üzerinden tanimladigi için, LLM'ler ile kod yazmama olayı onlara bir tehtid gibi geliyor ve icgüdüsel olarak karşı koyuyorlar. Lakin durum çok farklı bir yere geldi, şöyle bir örnek üzerinden açıklayayım. Artık hiç kimse bir derleyicinin ürettigi assembler koduna bakip, bu olmamis, ben daha iyi yaparım demiyor. Herkes artık gözü kapali derleyicilere güveniyor. İşte yeni cagin derleyicileri LLM'ler.

Kod yazmak yazilimcilik mesleginin zaten çok küçük bir bölümü idi. Ana mesele her daim müsteriyi anlamak ve onun istediği ürünü ortaya koyabilmektir. Bunu kod yazarak değil, çok büyük oranda çerçeveyi tayin ederek yapıyoruz. Bu belli olduktan sonra, oturup kodu yazmak artık marifet değil. Artık bu sıkıntı da ortadan kalktigina göre, artık tamamen müsteri ve onun gereksinimlerine odaklanabiliriz.

Yapay Zeka Yolculuğumun Kısa Hikayesi

Kaynak: https://kurumsaljava.com/2026/06/10/yazay-zeka-yolculugumun-kisa-hikayesi/

Uzun bir zaman önce şu an üzerinde çalıştığım ürünü geliştirmeye başladım. Ilk zamanlarda Github Copilot IntelliJ plugini ile auto complete yaparak ilerledim. O zamanlar bugünkü anlamda agentik bir yazılım modeli mümkün değildi.

Modeller iyilesmeye basladiktan sonra, takriben 1.5 sene önce Claude Sonnet 4.5 ile yazilimi tamamen yapay zekaya devreme sürecim başladı. Bu modellerin yetersiz kalmaya basladigini anladiktan sonra Opus 4.5 e geçtim ve çok hızlı bir şekilde üzerinde çalıştığım uygulamayı geliştirmeye devam ettim.

Geçen senenin agustos ayinda Opus 4.6 kullanmaya başladım. Bu bendeki yazılım konusundaki fikirsel kirilimin baslangici oldu. O noktadan itibaren ışık hizi ile geliştirmeye devam ettim. Normal sartlarda aylar ya da yillarimi alacak olan modülleri sadece haftalar içinde hayata gecirmek mümkün hale geldi. Zamanim olmadığı için ürüne dahil etmek istemedigim ve gelistirilmeleri karmaşık ürünleri bile pipelinea alarak, devam ettim.

Çok uzun bir süre Github Copilot CLI ile çalıştım. 3-4 ay öncesine kadar herşey hızlı bir şekilde ilerlerken, degisen kullanim ve fiyat politikaları ile bu araci kullanamaz hale geldim. Uzun bir süre buna dayanmaya çalıştım. Paramla rezil oldum. Haftalar süren rate limitler yedim. Opus 4.7 gelmesi ve Opus 4.6 nin devre dışı birakilmasi ile isler daha da kötülesmeye ve ilk baslardaki heyacanim kaybolmaya başladı.

Iki hafta önce Copilot kullanmayı sonlandirdim ve Claude code ve max20 aboneligine geçtim. Opus 4.8 ile orada tanistim ve Opus 4.6 dan çok daha iyi bir deneyim yasamaya başladım.

Bugün itibarei ile Fable 5 kullanmaya başladım ve ilk feature implementasyonunu yaptirdim. Gerçekten de tadindan yenmiyor. Bu hizla düşündüğümden daha kısa bir zamanda ürünü tamamlamis olacagim.

Ben bu satirlari yazerken Fable arka planda yeni bir feature implemente ediyor :)

Gereksinim Analizi

Kaynak: https://kurumsaljava.com/2026/06/10/gerensinim-analizi/

Bir feature implementasyonu öncesinde, ait olduğu alana ve hangi özelliklere sahip olması gerektiğini daha iyi anlayabilmek için gereksinim analizi ile başlıyorum. Bunun gerçek muhattabi aslında müsteri. Lakin ben belli bir piyasa için bir ürün gelistiriyorum ve bir müsteriye dogrudan erisimim olmadığı için piyasa arastirmasi, ürün kiyaslamasi ve yapay zeka yardımı ile gereksinim analizi yaparak ilerlemek zorundayim.

Burada zaman içinde bir çalışma sablonu olustu. Önü sizinle paylaşmak istiyorum.

Claude modelleri teknik analiz konusunda çok iyiler. Özellikle mevcut kodu inceleme ve gerekli degisikliklerin nasıl yapılması gerektiğine dair nokta atisi analizler yapıyorlar. Lakin bir alana ait gereksinim analizinde Chatgpt'nin daha iyi olduğunu fark ettim. Onunla karsilikli dialog halinde bir konu üzerinde çalışmak çok daha kolay. Bu yüzden tüm gereksinim analizini onunla yapıyorum ve çıkan analizi Fable 5 ile paylaşıyorum. Eğer gereksinim tam anlamıyla ve detaylı olarak analiz edildi ise, Fable 5 bu analizi çok hızlı bir şekilde, otonom olarak eksiksiz olarak koda dönüstürebiliyor.

Implementasyon sonunda oluşan kodu Chatgpt ile paylasarak, kodun yapılan analizi ne kadar tatmin ettigini soruyorum. Bu şekilde ping-pong vari feature implementasyonu tamamliyorum.

Artık Kod Yazmaya Gerek yok

Kaynak: https://kurumsaljava.com/2026/06/11/artik-kod-yazmaya-gerek-yok/

Artık kod yazmamiza gerek yok. Son 10 paylasimima bakarsaniz, yazdiklarimin ana temasi bu noktaya işaret etmektedir. Örnek olarak derleyicileri verdim. Bu konuyu biraz daha irdelemek istiyorum.

Bir derleyici ve bir LLM aynı şeyler degiller, ama prensiplte çok benzer çalışıyorlar. Ortak özellikleri ve farkliliklarina programcı perspektifinden inceleyelim.

Programcının yazılım konusundaki rolünün başka bir alana kaydigi konusunda hemfikir oldugumuzu düşünüyorum. Bu artık hiç kod yazmamayi ya da çok az. kod yazmayı gerektiren bir değişim. Bana göre artık kod yazmaya gerek yok. Bazilarina göre var.

Derleyici-LLM analojisini yakindan inceleyim. LLM öncesi mekanizma şu şekilde isliyordu: Programcı -> Java kodu -> Derleyici -> Makina kodu. LLM'ler ile bu şu şekilde değişti: Programcı -> Prompt -> LLM -> Java Code -> Derleyici -> Makina kodu. Buna göre LLM artık yeni derleyici seviyesi.

Hem derleyici hem de LLM örneginde soyutluk seviyesi değişti. Derleyiciler ile makina kodu yazma zorunlulugu ortadan kalkti. Yanı programcılar artık makina kodu yazmaz oldular. Bunun için bir yüksek seviyeye gecerek, isteklerine Java ya da C gibi yüksek bir dilde ifade etmeye başladılar. Aynı şey yapay zeka kullanımında da gerceklesti. Soyutluk seviyesi Java ya da C ile kod yazmaktan, ne yapılacağını LLM'lere söyleme seviyesine gecti.

Enformatik tarihi aslında sürekli soyutluk seviyesinin yükselmesinin toplamidir.

Örneğin 1960 larin bir Assembler programcısı günümüz programcisina büyük bir ihtimalle şunu söylerdi: "siz modern programcılar artık programlama yapmiyorsunuz. Sadece mantigi tarif ediyorsunuz".

1980 lerin bir C programcısı şunu söylerdi: "Java programcıları donanimdan bihaber."

Günümüzde ise LLM'ler ile artık kod yazmiyoruz. Yapısal olarak bunların hepsi aynı kapiya çıkıyor.

Tekrar derleyiciye dönelim. LLM ile derleyici arasındaki en önemli fark, derleyicilerin kod derlerken semantik bir garanti vermeleridir. Bu yüzden çıktıları deterministiktir ve hep aynı sonuclari verirler. Ama bir LLM için bunu söylemek mümkün değildir. LLM bir sonraki token'in ne olduğunu hesaplar ve bu istatistiksel hesaplama her defasinda başka bir sonuç verebilir. Bu yüzden LLM

bünyesinde formal bir dogrulama mantigi yoktur. Sonuç deterministik değildir. Aynı prompt değişik kod üretebilir. Ama çıkan iki değişik sonuç yine de doğru olabilir. bu kısmı biraz acalim.

Bir Java programcısı hayal edelim. Bu programcidan bir Spring boot uygulamasi gelistirmesini isteyelim. Bu programcı 3 ay arayla aynı uygulamayı iki sefer sıfırdan programlasin. Sizce bir derleyici gibi aynı kodu yazabilir mi? Ama emin olabileceginiz şey ne olacaktır? Uygulama çok değişik yapilandirilmis olsa bile, istediğimiz şekilde çalışıyor olacaktır. LLM de zaten aynı şeyi yapmaktadir. Her seferinde aynı prompt ile başka bir kodun ortaya cikmasi sorun teskil etmemektedir. Önemli olan semantik transformasyondur. Semantik anlamda kod benzer yapıda olduğu sürecek, sonuç istediğimiz şekilde olacaktır.

Günümüzde LLM'lere karşı kullanilan deterministik olmadiklari argümani geçerli bir argüman değildir. Çalışan bir kod için LLM'in deterministik olmayisi önem arz etmemektedir. Ona kalirsa bir programcı da deterministik değildir, ama her deneyisinde çalışan bir kod üretebilmektedir.

Yine de derleyici ve LLM arasında mutlak bir farklilik vardır. Derleyici kendi işini yapabilmesi için kodun sekillendirilmis olması gerekmektedir. Yanı derleyici çözüm üretemez. Ama LLM ona verilen bir prompt ile bir çözüm üretebilmektedir. Bu yüzden LLM sadece bir derleyici değildir, bir programcı gibi çözüm gelistiren birseydir.

Teknoloji Ötesi

Kaynak: `https://kurumsaljava.com/2026/06/11/teknoloji-otesi/`

Kusuruma bakmayin arkadaşlar, ama büyük bir değişim icindeyiz yazılım camiasi olarak ve topyekün teknolojinin arkadasindaki epistemolojik sorulara yönelmemiz gerekiyor, çünkü mevzu artık teknoloji, onun kullanımı ve bu konuda ne kadar iyi oldugunuz değil. Cevap bulmamiz gereken sorular var. Bunlar:

- Uzmanlik nedir?
- Ben artık bir programcı olarak kalabilir miyim?
- Rolüm ne olacak?
- Alan bilgisi mi önemlidir, kod bilgi mi?
- Yapay zeka çağında programcı olarak nerede olacagim?

Ben sürekli bu konulara deginiyorum, deginmeye devam edeceğim, çünkü bir cisim yaklaşıyor ve kendim için geçerli olmasa da sektör adin hazırlık yapmak gerekiyor. Nacizane tavsiyem mümkün mertebe hızlı bir şekilde bu konularda fikir yürütmeye baslamak olurdu.

Yazılım Artık Epistemolojidir

Kaynak: https://kurumsaljava.com/2026/06/11/yazilim-artik-epistemolojidir/

Yazlilm bir ontoloji olmaktan çıkıp, yapay zeka ile epistemoloji olma yoluna girdi. Teknoloji ve varligini konusurken (nasıl programlanir, nasıl çalışır, framework nedir vs), onun yerine birdenbire bilinc nedir (skynet), LLM çağında bilmek nedir, uzmanlik nedir gibi konuları konusuyoruz artık.

Bu geçiş artık ne yazık ki kolay değil. Bu yazımda teknolojik fay hatlarindan bahsetmistim:

> Zaman Eksenindeki Teknolojik Fay Hatlarının Programcılar Üzerindeki Etkileri

Bu artık teknollojik bir fay hatti değil, başka bir yasam formuna evrilme. Bu konudaki düsüncelerimi sonraki yazilarimda açıklamaya devam edeceğim. README.md

İşin Özü

Kaynak: https://kurumsaljava.com/2026/06/11/isin-ozu/

Şimdi kısa bir yazılım tarihine göz atalım.

Bit ve bytelar ile başlayan şey daha sonra makina kodu, assembly, c, java, çatılar, dsl, no-code ve llm olarak devam etti. Biz şu anda bu zincirin en tepesinde duruyoruz. Yanı programcılığa en soyut seviyeden bakıyoruz.

Her seviye bizim program yazış tarzımız üzerinde etkili oldu. Çalışma tarzımız değişti, ama değişmeyen bir öz var. Kod yazıyoruz ama onun da şekli, şemali değişiyor. Yanı kod yazmak işin özü değil. Değişmeyen o öz ne?

Programlama aslında kod yazmak değil, niyet ifade etmektir.

Kod yazma işini LLM'e bırakmak, işin özünden uzaklaştığımız anlamına gelmez. Aksine LLM bizi bu öze daha da yakınlaştırdı, çünkü artık alis harikalar diyarında vari istediğimiz herşeyi yapabilir hale geldik. Niyetimizi bir prompt ile çok daha net ifade etme şansımız var. Bir DSL (domain specific language) olan prompt insanın mantık ve doğasına en yakın olan bir yapı ve o bahsettiğim özde kalmanın en kolay yolu.

O yüzden asıl mesele kod yazmak mı yazmamak mı değil, işin özünden devam etmek mi, etmemek mi dir.

Not: Benim yazılarım daha çok işin epistemoloji tarafına dönüyor olacak. Szin için sıkıcı olmaya başladı ise, haber verin, bu konuda yazmayı bırakacağım :)

Yeni Soyutluk Seviyesi

Kaynak: https://kurumsaljava.com/2026/06/11/yeni-soyutluk-seviyesi/

Soru net olarak şu:

LLM'ler programlamanın özünü mü değiştiyor yoksa bir sonraki soyutluk katmanına hazırlık mı?

Enformatik tarihine baktığınızda, bu sorunun cevabını net olarak görüyoruz.

Bugünkü çalışma modeli:

Müsteri -> Gereksinim -> Programcı -> Prompt -> LLM -> Kod

Bir üstteki soyutluk seviyesi

Müsteri -> AI -> Ürün

Yanı ne olacak? Yeni soyutlama ya da sentez yapıldığında, alt katmanların üstü örtülür ve görünmez hale gelirler, çünkü onlara gerek kalmamıştır.

Programcılık diye bir meslek kalmayacak. Doğal dili anlayan yapay zeka müşterisi ile küçük bir sohbet sonrasında istediği ürünü ortaya koyacak.

Yapay Zeka İle Nasıl Çalışıyorum

Kaynak: https://kurumsaljava.com/2026/06/12/yapay-zeka-ile-nasil-calisiyorum/

Yapay zeka ile degisen çalışma tarzimi örnek olabilmesi acisindan sizinle paylaşmak istiyorum.

Artık IntelliJ / Android Studio uygulamalarini yapay zekanın yaptığı değişiklikleri takip etmek için kullanıyorum. Çoğu zaman commit penceresindeyim ve üzerinde değişiklik yapılan ya da yeni eklenen dosyaları, ne yapıldığını anlamak için editörde acarak, inceliyorum.

Bir prompt ile aldığım netice tatmin edici ise, hemen bir commit ile workspace i temizliyorum, çünkü bir sonraki prompt hataliysa ve geri alinmasi gerekiyorsa, mevcut çalışan koddan ayirt edilemeyecegi için tehlikeli bir durum olusuyor. Bunun önüne gecmek ve yapılan işlemleri birbirlerinden temiz bir şekilde ayirt edebilmek için sıkça commit yapıyorum. Feature tamamlandiginda git squash ile tek bir commite çevirip, pushluyorum.

Yapay zeka ile interaksiyonum sadece konsol üzerinden gerceklesiyor. Claude code cli kulaniyorum. Her uygulama katmani için ayrı bir workspace kullanıyorum ve her katman için bir cli çalışıyor.

Chatgpt desktop / web uygulama geliştirmek için kullandığım ikinci yapay zeka araci. Önü gereksinim analizi için kullanıyorum. Oradan çıkan analizleri Fable 5 ile paylasip, implementasyonu yaptırıyorum.

Artık kod ve test yazmıyorum. Fable 5 in entegrasyon testleri ve birim testleri yazmasini istiyorum. Bunlara güvenmedigim için onay / kabul testleri için onay/kabul kriterlerini ihtiva eden specler hazirliyorum. Fable 5 bunlardan test kodu oluşturup, kosturabiliyor. Böylece uygulamanın blackbox olarak test edilmesi ve onay/kabul kriterlerinin tatmin edilmis olmaları test baglaminda yeterli oluyor.

Yeni bir projeye basladigimda öncelikli bir programlama dil secimim olmuyor. Genelde Java, Typescript, Dart gibi dilleri kullanıyorum. Lakin çalıştığım alana göre bu değişiklik gösterebiliyor. Örneğin remote control amacıyla cihazlar üzerinde çalışan agent uygulamalari için Go dilini kullanmıştım. Artık poligot olan ben degilim, yapay zekanın kendisi.

Bir problem ile karşılaştığımda sorunu kendim çözmeye calismiyorum, debug yapmak gibi. Burada yine yapay zekaya gerekli talimatlari, loglari, oluşan veri setini vererek, problemi analiz etmesini ve çözmesini istiyorum.

Yapay zekanın yaptığı her analiz sonunda çözümü ihtiva eden bir dosya oluştur-

masını istiyorum. Bu dosyalar belli bir dizinde toplaniyor ve kodun ve uygulamanın hafizasi haline dönüşüyor.

Yeni bir uygulama geliştirmek istedigimde, önce ilk modülleri elden programlayip, referans olacak mimari, modüler yapı ve teknik çerçeveyi olustuyorum. Yapay zeka devreye girdiginde bu yapıları birebir kopyalamaya başlıyor. Bu şekilde mimariyi ve kod yapisini belli bir cercevede tutmak mümkün oluyor.

Tamamen biraktigim aliskanliklarim:

- Kod / test yazmak
- Ide içinde ya da gradle ile testleri kosturmak
- Test driven development yapmak
- Debug yapmak
- Refactoring yapmak

Ben artık tamamen AI first güdümlü çalışıyorum. Bu benim işin teknik tarafına değil, gereksinim analizi kismina odaklanmami mümkün kiliyor.

Kodun yapisina ve kalitesine dolaylı olarak, olusturdugum çerçeve dahilinde etki etmis oluyorum. Bu yüzden yapay zekanın oluşturduğu kod hakkında endisem olmuyor. Özetle kodun kalitesi, yapısı ve ne durumda olduğu konuları ile ilgilenmiyorum. Beni ilgilendiren tek şey onay / kabul kriterlerinin uygulama bünyesinde doğru bir şekilde vücut bulmus olmaları. Oluşan uygulama benim için bir blackbox ve içinde ne dolaplar döndügü, benim istediğim şekilde calistigi sürece beni hiç ilgilendirmiyor.

Gereksinim Analizi

Kaynak: https://kurumsaljava.com/2026/06/12/gereksinim-analizi/

Bir ürün geliştirmek için muhattap alinmasi gereken tek şahıs müsteridir. En ideal sartlarda müsteri SIKlikla yazılım ekibi ile bir araya gelir ve soruları cevaplar. Çoğu zaman aslında müsterinin de ne istedigini bilmedigi ortaya cikar, ama yine de ürüne dair bir vizyon geliştirmek ve gerekli ürünü ortaya koymak bu şekilde kolaylasir.

Müsterinin yazılım ekibi ile bir araya gelemeyecegi durumlarda bir vekile (customer proxy) ihtiyaç duyulur. Bir şekilde gereksinimlerin ekip içindeki yazılımcılar tarafından analiz edilmesi ve koda dönüstürülebilir hale getirilmesi gerekmektedir. Müsterinin vekili bu görevi üstlenir ve yazılım ekibi ve müsteri arasında köprü vazifesi görür.

Bu bahsettiğim iki durum, ya müsteri ya da vekilinin yazılım ekibi ile çalışıyor olması, olması gereken en sağlıklı yöntemdir. Ama çok ender rastlanan bir türdür. Genelde bir gereksinim analiz mühendisi kafasina göre gereksinim analizi yapar ve bunları yazılım ekibine verir. Nasıl bir sonuç ciktigina da mutlaka sahit olmuşsunuzdur.

Benim durumumda ortada ne müsteri var ne de vekili. Gereksinim analizi işini de başka birisine devretmek istemezdim. Bu yüzden gereksinim analizi için Chatgpt yi kullanıyorum. Chatgpt bir nevi customer proxy rolüne geçiyor ve alan için gerekli analizi detaylı bir şekilde yapabiliyor.

Bu şekilde karmaşık bir alana ait gereksinimlerin tespitini, veri yapilarini, gerekli sürecleri ve alan analizlerini yaptirmak çok daha kolaylasiyor. Buradan çıkan analizler yardımı ile backend ve frontend için gerekli kodu Fable 5 kolay bir şekilde olusturabiliyor. Artık kod yazmaya gerek kalmadı diyerek bu süreci kastetmis oluyorum.

Fable ile Chatgpt Nasıl Beraber Çalışırlar?

Kaynak: https://kurumsaljava.com/2026/06/13/fable-ile-chatgpt-nasil-beraber-calisirlar/

Bilgisayarimda claude code fable ve chatgpt desktop sürümünü birbirleri ile birlikte calisacak şekilde kullanıyorum. Chatgpt analiz konusunda, fable ile teknik implementasyon tarafında daha iyi. Aklima gelen ilk şey doğal olarak, benim aradan cekilerek, birbirleri ile sohbet halinde benim taleplerimi yerine getirebilip, getiremeyecekleri oluyor. Bu dogrudan mümkün değil, ama dolaylı olarak yapilabiliyor.

Birlikte calisabilmeleri için sagladiklarim:

- Ikisi de kodun yer aldigi workspace alanini dogrudan erisebiliyor. Chatgpt git aracılığı ile yapılan tüm degisikleri takip edebiliyor ve analizlerinde güncel kod durumunu kullanabiliyor.
- Fable işini bitirdikten sonra bir session report oluşturmasını istiyorum. Bunu chatgpt ile paylaşıyorum ve gerekli analiz buradan da yürüyebiliyor.
- Ortak CLAUDE.md dosyasini kullanabiliyorlar. Chatgpt bu dosyada yer alan temel prensipler doğrultusunda kendi analizlerini sekillendirebiliyor.

Bu saymis olduğum konular iki yapay zeka sisteminin birlikte calismalari için yeterli oluyor.

Henüz uygulamadiklarim:

- MCP üzerinden ikisinin arasında baglanti kurulabilir
- Ajan orkestrasyonu üzerinden isbirligi yapabilirler

Adim adim insansiz yazılıma doğru ilerliyoruz.

Fable 5 Masal Oldu

Kaynak: https://kurumsaljava.com/2026/06/13/fable-5-masal-oldu/

Birisi Fable 5 i hacklemis ve Mythos özelliklerini ortaya cikarmis. Claude 1000 saat test ettik, herşey güvenli diyordu :)

`https://lnkd.in/eHpgzN2K`

Bende şu an Fable 5 çalışıyor ama icerde Opus 4.8 e routing yapiyor olabilir. Zaten güvenlik söz konusu olunca default Opus 4.8 e yönlendiriyordu promptları.

Kac gün oldu Fable 5 cikali? 4 gün, 5 gün? Ben ciktigini duydugum ilk saatten beri Fable 5 kullanıyorum. GPT, Opus final onun yaninda hikaye. Fable gerçekten başka bir seviye.

Fable 5 ile yazilm yapmak ismine yarasir bir şekilde masal gibiydi. Hiç kodun yazilmadigi, sadece gereksinimlerin olduğu bir masal dünyasi. Ne yazık ki bitti.

Opus 4.8 isimizi görüyor, sıkıntı yok.

Adak falan mi adasak acaba Fable 5 için, belki daha hızlı geri gelir? Ben klavye ve faremi adak vermeye hazirim, yeter ki geri gelsin ve masala kaldigimiz yerden devam edelim.

Loop Engineering

Kaynak: https://kurumsaljava.com/2026/06/13/loop-engineering/

Ok, time for the next mind shift.

Artık prompt engineering değil, onun yerine loop enginering yapıyorum.

Ikisinin arasındaki fark ne?

Prompt engineering türünde yapay zeka ile sürekli interaksiyon halindesiniz. Prompt veriyorsunuz, o birşeyler yapiyor, siz sonuca bakiyorsunuz, düzeltmek istediklerinizi için yeniden bir prompt yaziyorsunuz ve bu döngü sürekli bu şekilde devam edip, gidiyor. Kısaca prompt engineering ile şu kodu yaz diyorsunuz. Github Copilot CLI ile bu şekilde calisiyordum.

Claude code ile durum değişti ve loop enginnering yapmak mümkün hale geldi benim için.

Loop engineering nedir?

Artık task bazli düsünmüyorum. Üzerinde çalışmak istediğim bir feature var ise, bunu bir bütün olarak Opus 4.8 de veriyorum ve o bir döngü içinde analizden testlere kadar gereken tüm adimlari kendi başına koşturuyor. Artık araya ben girmiyorum. Prompt engineering yaparken sürekli araya girmem gerekiyordu ve bu yüzden karmaşık uygulama özelliklerini (feature) implemente etmek daha uzun zaman aliyordu. Loop engineering sayesinde yapay zeka ile çalışma tarzim tamamen değişmiş oldu.

Pratikte bu loop nasıl olusturuluyor?

Bir ikipte 3 kişiyiz.

- Ben: Product owner ve alan için karar verici
- Gpt: Mimar, prompt designer ve kodu review yapan arkadasimiz
- Opus 4.8: Multiagent programcimiz, yanı programcilarimiz

Ilk adimda üzerinde callismak istediğim uygulama özelliğini (feature) seciyorum ve Gpt'ye şöyle bir prompt giriyorum:

> Bu yeni özelligi analiz et. Opus için bana bir loop-plani olustur. Bu plan model, backend, frontend, testler, rizikolar ve onay/kabul kriterlerini ihtiva etmeli.

Çıkan prompt 4,5 loop paragrafi ihtiva eder ve Opus bunları adim, adim imp-

lemente edecektir. Opus işini bitirdikten sonra tekrar Gtp'ye dönerek, Opus'un yaptıklarını analiz etmesini istiyorum. Buradan çıkan review dokümanini tekrar Opus ile paylaşıyorum ve eksikleri tamamlamasini istiyorum.

Benim buradaki tek rolüm neyin yapılacağını yanı neyi tayin etmek, nasıl yapilacagina Gpt ve Opus karar veriyorlar. Bunu gerceklestirmek için hem Gpt ve Opus arasında bir loop oluşturuyorum hem de Opus'un kendi loop mekanizmasinda istediğim sonucu cikarmasi için gerekli yönlendirmeyi Gpt sayesinde yapmis oluyorum.

Bir sonraki step tamamen aradan cekip, şunu yapin demek olacak. Bunun için Gpt ve Opus'u birlikte calisabilecek şekilde entegre etmek gerekiyor, yanı Gpt Opus'a gerekli loop promptları verebilmeli. Bu Jira üzerinden ticket olusturularak da çözülebilir. Bu konu üzerinde çalışacağım. Gelismelerden sizi haberdar ederim :)

Yapay zeka hakkında yazilarima buradan ulasabilirsiniz:

Yapay Zeka Konulu Yazılar

Hala Prompt Engineering mi Yapıyorsunuz?

Kaynak: https://kurumsaljava.com/2026/06/13/hala-prompt-engineering-mi-yapiyorsunz/

Komple bir envanter yönetimi sistemini (dashboard, envanter kalemleru, depolar, hareketler, depo transferleri, sayımlar, reçeteler, sipariş önerileri, satın alma siparişleri, mal kabulleri, tedarikçiler, raporlar bölümleri; angular, spring boot backend, sayısız yeni api, dto, mapper, service, entity sınıfları, liquibase scriptleri, unit ve entegrasyon testleri ile) loop engineering ile 3 gün içinde tamamladım. Kendin kodu yazmaya kalsaydım, muhtemelen minimum 3 ay uğraşırdım. Büyük bir ihtimalle daha uzun bile sürebilirdi. Buna paralel sayım yapmak ve depolarda mal kabulü için bir Flutter mobil uygulamasına başladım. O da yarın tamamlanmış olacak.

Ve…

Bir satır kod bile yazmadım!

Hala prompt engineering mi yapıyorsunuz?

Yapay Zeka Araç Kullanımı Nasıl Evrildi?

Kaynak: https://kurumsaljava.com/2026/06/13/yapay-zeka-arac-kullanimi-nasl-evrildi/

Yapay zeka araçları kullanımının zaman içinde bende nasıl değiştiğini kısaca paylaşayım.

- 2022 başlarında IIntelliJ IDEA bünyesinde Github Copilot plugini kullanmaya başladım. O zamanlar hatırladığım kadarıyla chat modu yoktu. Sadece autocomplete yapma özelligi ile ben kod yazarken bana destek oluyordu. Bunu ilk gördüğümde, yolculuğun nereye gideceğini anlamıştım.
- 2023 + 2024 de plugine chat özelligi eklendi ve bugün bildigimiz şekli ile prompt engineering yapmaya başladım.
- 2025 de Github copilot cli geldi ve ben tamemen IntelliJ / Android studio bünyeinde çalışmayı bıraktım ve konsola geçtim. Birkaç hafta öncesine kadar da prompt engineering yapmaya devam ettim. Burada birden fazla ajan ile çalışmanın nasıl olduguna dair ilk deneyimlerini yapmaya başladım.
- Birka hafta önce copilot'in kullanim ve fiyat politikaları radikal bir şekilde değiştiği için tamamen claude code'a geçtim.
- Bu zamandan beri Claude code ve Gpt ile loop engineering yapıyorum.

Loop enginering terimi aslında benim bugün dikkatimi çeken bir konuya dönüştü. Ben aslında haftalardır loop engineering yapıyormuşum, bugün onun farkına varmış oldum. Loop engineering ile uygulama özelliklerini (feature) tek sürümde bitirmek daha kolay bir hale geldi.

Bundan sonra yapay zeka araçları kullanımı nereye evrilecek dersiniz?

Küçük bir spoiler vereyim: ürün gelistirmenin insansiz yapıldığını hayal edin.

Intent Based Programming ve Intent Specific Language (ISL)

Kaynak: https://kurumsaljava.com/2026/06/14/intent-based-programming-ve-intent-specific-language

Eskiden programcı olarak nasıl soruna cevap arardım. Bu durum tamamen değişmiş durumda. Artık ne sorusunun peşindeyim. Programlama tarihi de LLM'ler ile birlikte nasıl sorusundan ne sorusuna doğru evrilen bir soyutlama tarihine dönüşüyor.

Günümüzde programlama paradigmalarının evrimine canli, canli sahit oluyoruz:

Imperatif -> Deklaratif -> Niyet Tabanli (intent based) Programlama

Niyet tabanli programlama ile ne kast ettigimi açıklayacağım.

Ben Java dilinde çok uzun bir dönem imperatif kod yazdım. Java 8 ile gelen Stream API ilk deklaratif kod yazma imkanı sagladi. Java 8 öncesi for, if, add yapısı Java 8 ile filter, map, collect e döndü. For döngüsü yerine filter yapisini kullanarak ne yapılması gerektiğini tayin ediyorum, ama nasıl yapılması gerektiğine JVM kendisi karar veriyor. Stream API ile gelen en büyük değişiklik bu oldu.

Yapay zeka ile bu çalışma tarzı yine bir üst soyutluk seviyesine taşınmış oldu. Artık promptlar ile "x modülünü yap, şunu desteklesin, bunu yapabilsin" yine ne yapılmasını gerektiğini tayin ediyorum, ama artık şunu filtrele, şu algoritmayı kullan bile demiyorum. Sadece niyet (intent) belirtiyorum. Bu da aslında yeni bir paradigmanın doğuşu: Intent Programming. Belki buna semantik programlama bile denebilir.

Prompt benim için bir DSL (domain specific language). DSL ile bir problem tanımlanır. Aynı şey prompt için de geçerli. Prompt doğal dil ile yazılmış bir DSL.

Bir üst soyutluk seviyesine saracak olursak, prompt aslında DSL değil, ISL (Intent Specific Language), çünkü DSL'de olduğu gibi syntax artık önemli değil, daha çok semantik içerik önemli.

Eskiden derleyiciler Java -> Bytecode -> Makina kodu şeklinde çalışıyorlardı. Şimdi ise niyet -> LLM -> Java, yanı LLM bir üst seviye derleyici gibi davranıyor. Kısaca LLM'lere prompt compiler diyebiliriz.

Eskiden yazılımcı nasıl yapıldığını bilen kişi idi. Yeni yazılımcı ise problemi en doğru şekilde modelleyen kişi olacak. Burada yine domain driven design öne çıkıyor, çünkü ne sorusuna cevap verebilmek için alanı çok iyi tanımak lazım.

Ben artık kod yazmıyorum. Sadece niyetimi ifade ediyorum. Kod insan ile makine arasındaki son ortak dil olmaktan çıkıp, yapay zekanın insanlar adina ürettigi bir ara temsile dönüşüyor. Şimdiye kadar gördügümüz tüm deklaratif programlama teknikleri son durak değildi. Onlar bizi prompt programlamaya hazırlamış oldular.

Yazar Hakkında

Özcan Acar, yazılım geliştirme, Java, çevik yazılım geliştirme, yazılım mimarisi ve yapay zeka destekli yazılım geliştirme konularında yazılar yazan bir yazılım geliştiricisidir.

Web siteleri:

- https://kurumsaljava.com
- https://pratikprogramci.com

Sosyal medya:

- X: https://x.com/oezcanacar
- Insta: https://www.instagram.com/oezcanacar