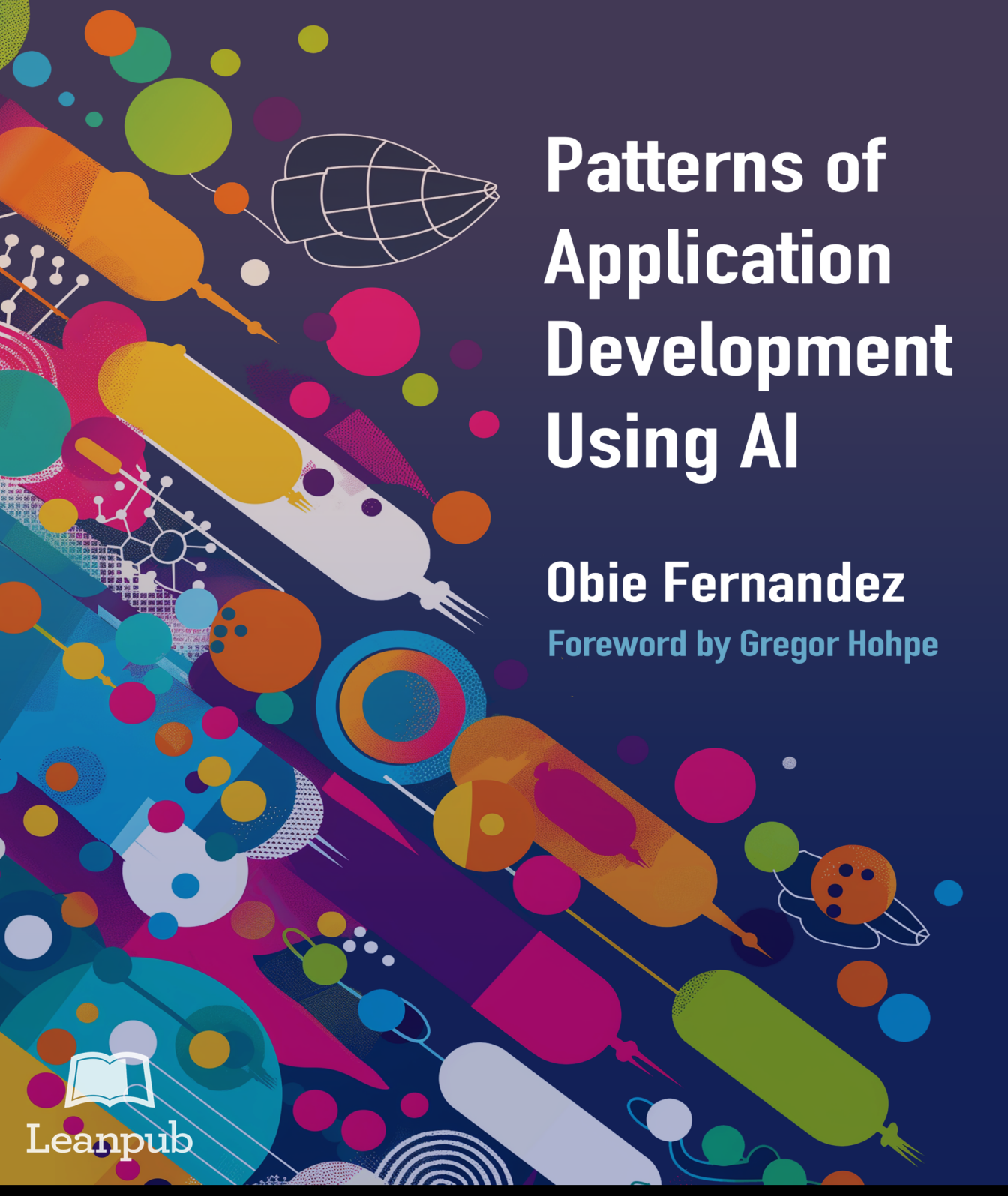


Norsk Utgave

Mønstre for Applikasjonsutvikling med KI (Norsk Utgave)

Obie Fernandez

Denne boken er til salgs på
http://leanpub.com/patterns-of-application-development-using-ai-nb

Denne versjonen ble publisert 2025-01-23

Dette er en Leanpub bok. Leanpub utdanner forfattere og utgivere med “Lean Publisering”-prosessen. Lean Publisering hjelper med å publisere arbeidsversjoner av en bok, bruker lette verktøy og mange iterasjoner for å få leserens tilbakemeldinger til du har den rette boken og bygger interesse mens du gjør det.

Tweet denne boken!

Hjelp Obie Fernandez ved å fortelle om denne boken på Twitter!

Den foreslåtte hashtagen av denne boken er: #poaduai.

Finn ut hva andre sier om boken ved å følge denne lenken til et søk på Twitter:

#poaduai

Til min tøffe dronning, min muse, mitt lys og min kjærlighet, Victoria

Også av Obie Fernandez

Patterns of Application Development Using AI

The Rails 8 Way

The Rails 7 Way

XML The Rails Way

Serverless

El Libro Principiante de Node

The Lean Enterprise

Innhald

Forord av Gregor Hohpe . . . . . **i**

Forord . . . . . **ii**

Om boken iii

Om kodeeksemplene iii

Hva jeg ikke dekker iii

Hvem denne boken er for iii

Bygge et Felles Vokabular iii

Bli Involvert iii

Anerkjennelser iii

Hva er greia med illustrasjonene? iv

Om Lean Publishing iv

Om forfatteren v

Introduksjon **1**

Tanker om programvarearkitektur 2

Hva er en stor språkmodell? 3

Forstå inferens 5

Tenke på ytelse 25

Eksperimentere med forskjellige LLM-modeller 27

Sammensatte AI-systemer 27

Del 1: Grunnleggende tilnærminger og teknikker 35

Innsnevre stien **36**

Latent rom: Ubegripelig stort 38

Hvordan Stien Blir "Innsnevret" 42

Råmodeller versus instruksjonsjusterte modeller 45

Prompt-utforming 52

Promptdestillering 68

Hva med finjustering? 74

Retrieval Augmented Generation (RAG) **76**

Hva er Retrieval Augmented Generation? 76

Hvordan fungerer RAG? 76

Hvorfor bruke RAG i applikasjonene dine? 76

Implementering av RAG i Din Applikasjon 76

Påstandsoppdeling 77

Praktiske eksempler på RAG 77

Intelligent spørringsoptimalisering (IQO) 78

Rerangering 78

RAG-vurdering (RAGAs) 78

Utfordringer og Fremtidsutsikter 80

Mangfold av arbeidere **82**

KI-arbeidere som uavhengige gjenbrukbare komponenter 83

Kontoadministrasjon 85

E-handelapplikasjoner 86

Helsetjenesteanvendelser 94

KI-arbeider som prosesshåndterer 97

Integrering av AI-Arbeidere i Applikasjonsarkitekturen Din 101

Sammenstillbarhet og orkestrering av AI-arbeidere 104
Kombinere tradisjonell NLP med LLMer 113

Verktøybruk 116
Hva er verktøybruk? 116
Potensialet i verktøybruk 118
Arbeidsflyten for verktøybruk 119
Beste praksis for verktøybruk 133
Sammensetting og Kjeding av Verktøy 137
Fremtidige Retninger 139

Strømmebehandling 142
Implementering av en ReplyStream 143
"Samtaleløkken" 149
Automatisk fortsettelse 151
Konklusjon 153

Selvhelbredende data 155
Praktisk casestudie: Reparering av ødelagt JSON 157
Hensyn og kontraindikasjoner 162

Kontekstuell innholdsgenerering 177
Personalisering 178
Produktivitet 180
Rask iterasjon og eksperimentering 182
AI-drevet lokalisering 184
Viktigheten av Brukertesting og Tilbakemelding 186

Generative UI 187
Generering av tekst for brukergrensesnitt 188
Definering av Generativ UI 197

Eksempel . . . 199
Skiftet til Resultatorientert Design . . . 201
Utfordringer og Hensyn . . . 203
Fremtidsutsikter og Muligheter . . . 204

Intelligent arbeidsflytorkestrering . . . 208
Forretningsmessig behov . . . 209
Sentrale fordeler . . . 210
Nøkkelmønstre . . . 210
Unntakshåndtering og gjenoppretting . . . 213
Implementering av Intelligent Arbeidsflytorkestrering i Praksis . . . 216
Overvåking og Logging . . . 231
Skalerbarhet og Ytelseshensyn . . . 235
Testing og validering av arbeidsflyter . . . 240

Del 2: Mønstrene . . . 248

Prompt-konstruksjon . . . 249
Tankerekke . . . 250
Modusveksling . . . 251
Rolletildeling . . . 252
Prompt Object . . . 253
Prompt Template . . . 254
Structured IO . . . 255
Prompt-kjeding . . . 256
Prompt-omskriver . . . 257
Responsbegrensning . . . 258
Spørringsanalysator . . . 259
Spørringsomskriver . . . 260
Ventriloquist . . . 261

Diskrete komponenter **262**
Predikat 263
API-fasade 264
Result Interpreter 266
Virtuell Maskin 267
Spesifikasjon og Testing 267

Human In The Loop (HITL) **269**
Overordnede mønstre 269
Eskalering 270
Tilbakemeldingssløyfe 271
Passiv informasjonsutstråling 272
Samarbeidende Beslutningstaking (CDM) 274
Kontinuerlig læring 275
Etiske hensyn 275
Teknologiske fremskritt og fremtidsutsikter 275

Intelligent feilhåndtering **277**
Tradisjonelle feilhåndteringstilnærminger 277
Kontekstuell feildiagnose 278
Intelligent feilrapportering 279
Prediktiv feilforebygging 280
Smart feilgjenoppretting 280
Personalisert feilkommunikasjon 281
Adaptiv feilhåndteringsarbeidsflyt 282

Kvalitetskontroll **283**
Eval 284
Sikkerhetsmekanisme 286
Guardrails og Evalueringer: To Sider av Samme Sak 286

Ordliste **288**
Ordliste 288

Index **293**

Forord av Gregor Hohpe

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Forord

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Om boken

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Om kodeeksemplene

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hva jeg ikke dekker

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvem denne boken er for

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Bygge et Felles Vokabular

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Bli Involvert

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Anerkjennelser

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hva er greia med illustrasjonene?

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

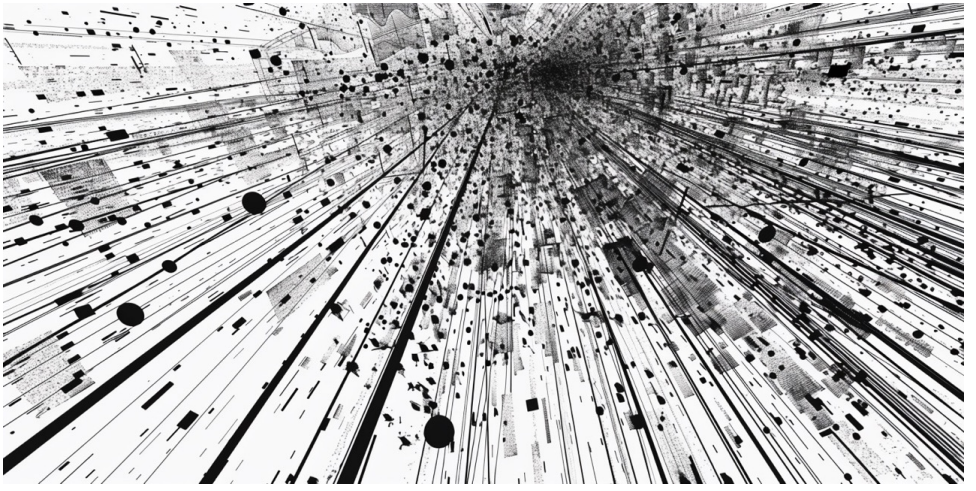
Om Lean Publishing

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Om forfatteren

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Introduksjon



Hvis du er ivrig etter å begynne å integrere store språkmodeller (LLM) i programmeringsprosjektene dine, kan du gjerne hoppe rett til mønstrene og kodeeksemplene som presenteres i senere kapitler. Men for å fullt ut verdsette kraften og potensialet i disse mønstrene, er det verdt å ta seg tid til å forstå den bredere sammenhengen og den helhetlige tilnærmingen de representerer.

Mønstrene er ikke bare en samling isolerte teknikker, men snarere et enhetlig rammeverk for å integrere AI i applikasjonene dine. Jeg bruker Ruby on Rails, men disse mønstrene burde fungere i stort sett alle andre programmeringsmiljøer. De tar for seg et bredt spekter av hensyn, fra datahåndtering og ytelsesoptimalisering til brukeropplevelse og sikkerhet, og gir et omfattende verktøysett for å forbedre tradisjonell programmeringspraksis med AI-funksjonalitet.

Hver kategori av mønstre tar for seg en spesifikk utfordring eller mulighet som oppstår når man inkorporerer AI-komponenter i applikasjonen din. Ved å forstå relasjonene

og synergiene mellom disse mønstrene, kan du ta informerte beslutninger om hvor og hvordan AI kan anvendes mest effektivt.

Mønstre er aldri forskrivende løsninger og bør ikke behandles som det. De er ment å være tilpasningsdyktige byggeklosser som bør skreddersys til de unike kravene og begrensningene i din egen applikasjon. Vellykket anvendelse av disse mønstrene (som alle andre innen programvareutvikling) avhenger av en dyp forståelse av problemdomenet, brukerbehov og den overordnede tekniske arkitekturen i prosjektet ditt.

Tanker om programvarearkitektur

Jeg begynte å programmere på 1980-tallet og var involvert i hackermiljøet, og mistet aldri min hackermentalitet, selv etter at jeg ble profesjonell programvareutvikler. Helt fra starten hadde jeg alltid en sunn skepsis til hvilken verdi programvarearkitekter i sine elfenbenstårn faktisk tilførte.

En av grunnene til at jeg personlig er så begeistret for endringene som denne kraftige nye bølgen av AI-teknologi bringer med seg, er dens innvirkning på det vi anser som *programvarearkitektur*-beslutninger. Den utfordrer tradisjonelle oppfatninger om hva som utgjør den "riktige" måten å designe og implementere programvareprosjekter på. Den utfordrer også om arkitektur fortsatt kan betraktes primært som *de delene av et system som er vanskelige å endre*, siden AI-forbedringer gjør det enklere enn noensinne å endre hvilken som helst del av prosjektet ditt, når som helst.

Kanskje vi går inn i toppårene for den "postmoderne" tilnærmingen til programvareutvikling. I denne sammenhengen refererer postmoderne til et fundamentalt skifte bort fra tradisjonelle paradigmer, der utviklere var ansvarlige for å skrive og vedlikeholde hver eneste kodelinje. I stedet omfavner den ideen om å delegere oppgaver, som datamanipulering, komplekse algoritmer og til og med hele deler av applikasjonslogikken, til tredjepartsbiblioteker og eksterne API-er. Dette

postmoderne skiftet representerer et betydelig avvik fra den konvensjonelle visdommen om å bygge applikasjoner fra bunnen av, og det utfordrer utviklere til å tenke nytt om sin rolle i utviklingsprosessen.

Jeg har alltid trodd at gode programmerere bare skriver den koden som er absolutt nødvendig å skrive, basert på læren fra Larry Wall og andre hackerluminærer som ham. Ved å minimere mengden skrevet kode kan vi bevege oss raskere, redusere overflatearealet for feil, forenkle vedlikehold og forbedre den generelle påliteligheten til applikasjonene våre. Mindre kode lar oss fokusere på kjernen i forretningslogikken og brukeropplevelsen, mens annet arbeid delegeres til andre tjenester.

Nå som AI-drevne systemer kan håndtere oppgaver som tidligere var forbeholdt menneskeskrevet kode, burde vi kunne være enda mer produktive og smidige, med et større fokus enn noensinne på å skape forretningsverdi og brukeropplevelse.

Selvfølgelig er det ulemper ved å delegere store deler av prosjektet ditt til AI-systemer, som potensielt tap av kontroll og behovet for robust overvåking og tilbakemeldingsmekanismer. Det er derfor det krever et nytt sett med ferdigheter og kunnskap, inkludert i det minste en grunnleggende forståelse av hvordan AI fungerer.

Hva er en stor språkmodell?

Store språkmodeller (LLM) er en type kunstig intelligens-modell som har fått betydelig oppmerksomhet de senere årene, helt siden lanseringen av GPT-3 av OpenAI i 2020. Store språkmodeller er designet for å behandle, forstå og generere menneskelig språk med bemerkelsesverdig nøyaktighet og flyt. I denne delen skal vi ta en kort titt på hvordan store språkmodeller fungerer og hvorfor de er godt egnet for å bygge intelligente systemkomponenter.

I kjernen er store språkmodeller basert på dyplæringsalgoritmer, spesifikt nevrale nettverk. Disse nettverkene består av sammenkoblede noder, eller nevroner, som behandler og overfører informasjon. Arkitekturen som foretrekkes for store

språkmodeller er ofte transformermodellen, som har vist seg å være svært effektiv i håndtering av sekvensielle data som tekst.

Transformermodeller er basert på oppmerksomhetsmekanismen og brukes hovedsakelig til oppgaver som involverer sekvensielle data, som naturlig språkbehandling. Transformere behandler inndata samtidig i stedet for sekvensielt, noe som gjør dem i stand til å fange opp langtrekkende avhengigheter mer effektivt. De har lag av oppmerksomhetsmekanismer som hjelper modellen med å fokusere på forskjellige deler av inndataene for å forstå kontekst og sammenhenger.

Treningsprosessen for store språkmodeller innebærer å eksponere modellen for enorme mengder tekstdata, som bøker, artikler, nettsider og kodelagre. Under treningen lærer modellen å gjenkjenne mønstre, relasjoner og strukturer i teksten. Den fanger opp de statistiske egenskapene til språket, som grammatiske regler, ordassosiasjoner og kontekstuelle betydninger.

En av nøkkelteknikkene som brukes i trening av store språkmodeller er ikke-overvåket læring. Dette betyr at modellen lærer fra dataene uten eksplisitt merking eller veiledning. Den oppdager mønstre og representasjoner på egen hånd ved å analysere samforekomsten av ord og fraser i treningsdataene. Dette gjør at store språkmodeller kan utvikle en dyp forståelse av språk og dets kompleksitet.

Et annet viktig aspekt ved store språkmodeller er deres evne til å håndtere *kontekst*. Når de behandler en tekst, vurderer store språkmodeller ikke bare de enkelte ordene, men også den omkringliggende konteksten. De tar hensyn til tidligere ord, setninger og til og med avsnitt for å forstå betydningen og intensjonen i teksten. Denne kontekstuelle forståelsen gjør store språkmodeller i stand til å generere sammenhengende og relevante svar. En av hovedmåtene vi evaluerer kapasiteten til en gitt språkmodell på, er ved å vurdere størrelsen på konteksten de kan ta hensyn til for å generere svar.

Når de er trent, kan store språkmodeller brukes til et bredt spekter av språkrelaterte oppgaver. De kan generere menneskelignende tekst, svare på spørsmål, oppsummere dokumenter, oversette språk og til og med skrive kode. Allsidigheten til store

språkmodeller gjør dem verdifulle for å bygge intelligente systemkomponenter som kan samhandle med brukere, behandle og analysere tekstdata, og generere meningsfullt innhold.

Ved å inkorporere store språkmodeller i applikasjonsarkitekturen kan du skape AI-komponenter som forstår og behandler brukerinndata, genererer dynamisk innhold og gir intelligente anbefalinger eller handlinger. Men å jobbe med store språkmodeller krever nøye vurdering av ressurskrav og ytelseskompromisser. Store språkmodeller er beregningsmessig intensive og kan kreve betydelig prosesseringskraft og minne (med andre ord, penger) for å operere. De fleste av oss må vurdere kostnadsimplikasjonene ved å integrere store språkmodeller i applikasjonene våre og handle deretter.

Forstå inferens

Inferens refererer til prosessen der en modell genererer prediksjoner eller output basert på nye, usette data. Det er fasen hvor den trente modellen brukes til å ta beslutninger eller generere tekst, bilder eller annet innhold som respons på brukerinndata.

Under treningsfasen lærer en AI-modell fra et stort datasett ved å justere parameterne sine for å minimere feilen i prediksjonene. Når modellen er trent, kan den anvende det den har lært på nye data. Inferens er hvordan modellen bruker sine lærte mønstre og kunnskap til å generere output.

For store språkmodeller innebærer inferens å ta imot en prompt eller inndatatekst og produsere et sammenhengende og kontekstuelt relevant svar, som en strøm av *tokens* (som vi skal snakke om snart). Dette kan være å svare på et spørsmål, fullføre en setning, generere en historie eller oversette tekst, blant mange andre oppgaver.

I motsetning til måten du og jeg tenker på, skjer en AI-modells "tenkning" via inferens i én tilstandsløs operasjon. Det vil si at dens tenkning er begrenset til genereringsprosessen. Den må bokstavelig talt tenke høyt, som om jeg stilte deg et spørsmål og bare godtok et svar fra deg i "stream of consciousness"-stil.

Store språkmodeller kommer i mange størrelser og varianter

Mens praktisk talt alle populære store språkmodeller er basert på den samme kjerne-transformerarkitekturen og trent på enorme tekstdatasett, kommer de i forskjellige størrelser og er finjustert for ulike formål. Størrelsen på en stor språkmodell, målt i antall parametere i dens nevrale nettverk, har stor innvirkning på dens kapabiliteter. Større modeller med flere parametere, som GPT-4, som det ryktes har 1 til 2 billioner parametere, er generelt mer kunnskapsrike og kapable enn mindre modeller. Imidlertid krever større modeller også mye mer datakraft for å kjøre, noe som betyr høyere kostnader når du bruker dem via API-kall.

For å gjøre store språkmodeller mer praktiske og skreddersydd for spesifikke bruksområder, blir basismodellene ofte finjustert på mer målrettede datasett. For eksempel kan en stor språkmodell trenes på et stort korpus av dialog for å spesialisere den for samtale-AI. Andre er trent på kode for å gi dem programmeringskunnskap. Det finnes til og med modeller som er spesielt trent for rollespill-lignende interaksjoner med brukere!

Gjenfinning vs Generative Modeller

I verden av store språkmodeller (LLM) finnes det to hovedtilnærminger for å generere svar: gjenfinningsbaserte modeller og generative modeller. Hver tilnærming har sine styrker og svakheter, og forståelse av forskjellene mellom dem kan hjelpe deg å velge riktig modell for ditt spesifikke bruksområde.

Gjenfinningsbaserte Modeller

Gjenfinningsbaserte modeller, også kjent som informasjonsgjenfinningsmodeller, genererer svar ved å søke gjennom en stor database med eksisterende tekst og velge de mest relevante avsnittene basert på inngangsforespørselen. Disse modellene genererer

ikke ny tekst fra bunnen av, men setter heller sammen utdrag fra databasen for å forme et sammenhengende svar.

En av hovedfordelene med gjenfinningsbaserte modeller er deres evne til å gi faktisk nøyaktig og oppdatert informasjon. Siden de er avhengige av en database med kuratert tekst, kan de hente relevant informasjon fra pålitelige kilder og presentere den for brukeren. Dette gjør dem godt egnet for applikasjoner som krever presise, faktabaserte svar, som spørsmål-og-svar-systemer eller kunnskapsbaser.

Gjenfinningsbaserte modeller har imidlertid noen begrensninger. De er bare så gode som databasen de søker gjennom, så kvaliteten og dekningen av databasen påvirker direkte modellens ytelse. I tillegg kan disse modellene streve med å generere sammenhengende og naturlig lydende svar, siden de er begrenset til teksten som er tilgjengelig i databasen.

Vi dekker ikke bruk av rene gjenfinningsmodeller i denne boken.

Generative Modeller

Generative modeller, på den annen side, skaper ny tekst fra bunnen av basert på mønstre og relasjoner de lærte under trening. Disse modellene bruker sin forståelse av språk til å generere nye svar som er skreddersydd for inngangsprompten.

Hovedstyrken til generative modeller er deres evne til å produsere kreativ, sammenhengende og kontekstuelt relevant tekst. De kan engasjere seg i åpne samtaler, generere historier og til og med skrive kode. Dette gjør dem ideelle for applikasjoner som krever mer åpne og dynamiske interaksjoner, som chatbots, innholdsproduksjon og kreative skriveassistenter.

Generative modeller kan imidlertid noen ganger produsere inkonsistent eller faktisk feil informasjon, siden de er avhengige av mønstre lært under trening i stedet for en kuratert database med fakta. De kan også være mer utsatt for skjevheter og hallusinasjoner, og generere tekst som er plausibel men ikke nødvendigvis sann.

Eksempler på generative LLM-er inkluderer OpenAIs GPT-serie (GPT-3, GPT-4) og Anthropics Claude.

Hybridmodeller

Flere kommersielt tilgjengelige LLM-er kombinerer både gjenfinnings- og generative tilnærminger i en hybridmodell. Disse modellene bruker gjenfinningstekniker for å finne relevant informasjon fra en database og bruker deretter generative teknikker for å syntetisere denne informasjonen til et sammenhengende svar.

Hybridmodeller tar sikte på å kombinere den faktiske nøyaktigheten til gjenfinningsbaserte modeller med språkgenereringsevnene til generative modeller. De kan gi mer pålitelig og oppdatert informasjon samtidig som de opprettholder evnen til å engasjere seg i åpne samtaler.

Når du velger mellom gjenfinningsbaserte og generative modeller, bør du vurdere de spesifikke kravene til applikasjonen din. Hvis hovedmålet er å gi nøyaktig, faktabasert informasjon, kan en gjenfinningsbasert modell være det beste valget. Hvis applikasjonen krever mer åpne og kreative interaksjoner, kan en generativ modell være mer egnet. Hybridmodeller tilbyr en balanse mellom de to tilnærmingene og kan være et godt valg for applikasjoner som krever både faktisk nøyaktighet og naturlig språkgenerering.

Til syvende og sist avhenger valget mellom gjenfinningsbaserte og generative modeller av det spesifikke brukstilfellet og avveiingene mellom nøyaktighet, kreativitet og fleksibilitet. Ved å forstå styrkene og begrensningene til hver tilnærming, kan du ta informerte beslutninger.

Instruksjonsfinjustering

Instruksjonsfinjustering er en undergruppe av konversasjonstrening som bruker menneskeskrevne prompts og svar for å gjøre LLM-er bedre på å forstå og følge instruksjoner. Her er noen enkle eksempler for å illustrere hvordan instruksjonsfinjustering på menneskeskrevne prompts og svar gjør LLM-er bedre på å forstå og følge instruksjoner:

Eksempel 1: Kundeservice

Instruksjon: "Gi et svar til en kunde som ber om refusjon."

Menneskeskrevet Prompt: "Jeg ønsker å be om refusjon for mitt nylige kjøp. Produktet oppfylte ikke mine forventninger."

Menneskeskrevet Svar: "Jeg beklager at produktet ikke oppfylte forventningene dine. For å behandle refusjonen din, vennligst oppgi ordrenummeret ditt og årsaken til returen. Vi vil gjennomgå forespørselen din og komme tilbake til deg innen 2-3 virkedager."

Eksempel 2: Pedagogisk Assistanse

Instruksjon: "Forklar konseptet fotosyntese til en 10-åring."

Menneskeskrevet prompt: "Kan du forklare hvordan planter lager maten sin?"

Menneskeskrevet svar: "Ja visst! Planter lager maten sin gjennom en prosess som kalles fotosyntese. De bruker sollys, vann og karbondioksid for å lage mat og oksygen. Bladene på planten har en spesiell del som kalles klorofyll som hjelper dem med å fange sollys."

Eksempel 3: Personlig assistent

> Instruksjon: “Planlegg et møte for neste tirsdag klokken 15:00.”
>
> Menneskeskrevet prompt: “Kan du sette opp et møte for tirsdag klokken 15:00?”
>
> Menneskeskrevet svar: “Ja visst, jeg har planlagt et møte for deg neste tirsdag klokken 15:00. Er det noe annet du trenger?”

Resultatet er et mangfoldig økosystem av LLM-er i forskjellige størrelser og med ulike spesialiteter. Mindre modeller i området 1-7 milliarder parametere gir gode generelle språkferdigheter samtidig som de er mer effektive å kjøre.

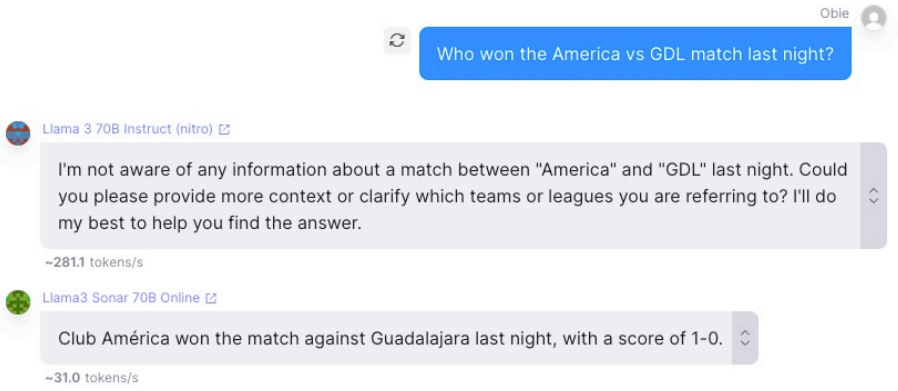
- Mistral 7B
- Llama 3 8B
- Gemma 7B

Mellomstore modeller på rundt 30-70 milliarder parametere tilbyr sterkere resonneringsevner og evne til å følge instruksjoner.

- Llama 3 70B
- Qwen2 70B
- Mixtral 8x22B

Når man velger en LLM som skal integreres i en applikasjon, må man balansere modellens evner mot praktiske faktorer som kostnad, latens, kontekstlengde og innholdsfiltrering. Mindre, instruksjonstilpassede modeller er ofte det beste valget for enklere språkoppgaver, mens de største modellene kan være nødvendige for kompleks resonnering eller analyse. Modellens treningsdata er også en viktig faktor, ettersom det bestemmer modellens kunnskapsgrense.

Enkelte modeller, som noen fra Perplexity, er koblet til sanntids informasjonskilder, slik at de i praksis ikke har noen kunnskapsgrense. Når du stiller dem spørsmål, kan de selvstendig bestemme seg for å gjøre nettsøk og hente vilkårlige nettsider for å generere et svar.

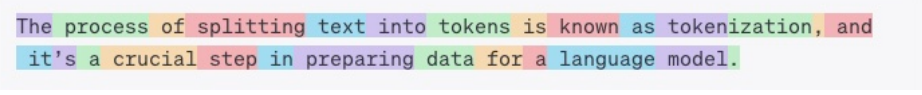


Figur 1. Llama3 med og uten nettilgang

Til syvende og sist finnes det ingen universalløsning når det gjelder LLM-er. Å forstå variasjonene i modellstørrelse, arkitektur og trening er nøkkelen til å velge riktig modell for et gitt bruksområde. Å eksperimentere med forskjellige modeller er den eneste praktiske måten å avdekke hvilke som gir best ytelse for oppgaven som skal løses.

Tokenisering: Å dele tekst inn i biter

Før en stor språkmodell kan behandle tekst, må teksten deles opp i mindre enheter som kalles *tokener*. Tokener kan være enkelte ord, deler av ord eller til og med enkelttegn. Prosessen med å dele tekst inn i tokener kalles tokenisering, og det er et avgjørende trinn i forberedelsen av data for en språkmodell.



Figur 2. Denne setningen inneholder 27 tokener

Forskjellige LLM-er bruker ulike tokeniseringsstrategier, som kan ha betydelig innvirkning på modellens ytelse og kapabiliteter. Noen vanlige tokenisatorer som

brukes av LLM-er inkluderer:

- **GPT (Byte Pair Encoding)**: GPT-tokenisatorer bruker en teknikk som kalles byte pair encoding (BPE) for å dele tekst inn i delordenheter. BPE slår iterativt sammen de mest frekvente parene av bytes i et tekstkorpus, og danner et vokabular av delordtokener. Dette gjør at tokenisatoren kan håndtere sjeldne og nye ord ved å dele dem opp i mer vanlige delorddeler. GPT-tokenisatorer brukes av modeller som GPT-3 og GPT-4.

- **Llama (SentencePiece)**: Llama-tokeniserere bruker SentencePiece-biblioteket, som er en ikke-overvåket teksttokeniserer og detokeniserer. SentencePiece behandler innteksten som en sekvens av Unicode-tegn og lærer et delordsvokabular basert på et treningskorpus. Den kan håndtere alle språk som kan kodes i Unicode, noe som gjør den godt egnet for flerspråklige modeller. Llama-tokeniserere brukes av modeller som Metas Llama og Alpaca.

- **SentencePiece (Unigram)**: SentencePiece-tokeniserere kan også bruke en annen algoritme kalt Unigram, som er basert på en delordregulariseringsteknikk. Unigram-tokenisering bestemmer det optimale delordsvokabularet basert på en unigram språkmodell, som tildeler sannsynligheter til individuelle delordsenheter. Denne tilnærmingen kan produsere mer semantisk meningsfulle delord sammenlignet med BPE. SentencePiece med Unigram brukes av modeller som Googles T5 og BERT.

- **Google Gemini (Multimodal Tokenisering)**: Google Gemini bruker et tokeniseringssystem designet for å håndtere ulike datatyper, inkludert tekst, bilder, lyd, videoer og kode. Denne multimodale kapasiteten lar Gemini behandle og integrere forskjellige former for informasjon. Spesielt har Google Gemini 1.5 Pro et kontekstvindu som kan håndtere millioner av tokens, mye større enn tidligere modeller. Dette omfattende kontekstvinduet gjør det mulig for modellen

å behandle en større kontekst, noe som potensielt fører til mer nøyaktige svar. Det er imidlertid viktig å merke seg at Geminis tokeniseringssystem er mye nærmere én token per tegn enn andre modeller. Dette betyr at den faktiske kostnaden ved å bruke Gemini-modeller kan være betydelig høyere enn forventet hvis du er vant til å bruke modeller som GPT, ettersom Googles prising er basert på tegn i stedet for tokens.

Valget av tokeniserer påvirker flere aspekter ved en LLM, inkludert:

- **Vokabularstørrelse**: Tokenisereren bestemmer størrelsen på modellens vokabular, som er settet av unike tokens den gjenkjenner. Et større, mer finkornet vokabular kan hjelpe modellen med å håndtere et bredere spekter av ord og fraser og til og med bli multimodal (i stand til å forstå og generere mer enn bare tekst), men det øker også modellens minnekrav og beregningsmessige kompleksitet.
- **Håndtering av sjeldne og ukjente ord**: Tokeniserere som bruker delordsenheter, som BPE og SentencePiece, kan bryte ned sjeldne og ukjente ord i mer vanlige delordsbiter. Dette lar modellen gjøre kvalifiserte gjetninger om betydningen av ord den ikke har sett før, basert på delordene de inneholder.
- **Flerspråklig støtte**: Tokeniserere som SentencePiece, som kan håndtere alle Unicode-kodbare språk, er godt egnet for flerspråklige modeller som må behandle tekst på flere språk.

Når man velger en LLM for en bestemt applikasjon, er det viktig å vurdere tokenisereren den bruker og hvor godt den samsvarer med de spesifikke språkbehandlingsbehovene for oppgaven. Tokenisereren kan ha betydelig innvirkning på modellens evne til å håndtere domenespesifikk terminologi, sjeldne ord og flerspråklig tekst.

Kontekststørrelse: Hvor mye informasjon kan en språkmodell bruke under inferens?

Når vi diskuterer språkmodeller, refererer kontekststørrelse til mengden tekst som en modell kan vurdere når den behandler eller genererer sine svar. Det er i hovedsak et mål på hvor mye informasjon modellen kan "huske" og bruke til å informere sine outputs (uttrykt i tokens). Kontekststørrelsen til en språkmodell kan ha betydelig innvirkning på dens kapabiliteter og typene oppgaver den kan utføre effektivt.

Hva er kontekststørrelse?

I tekniske termer bestemmes kontekststørrelsen av antall tokens (ord eller orddeler) som en språkmodell kan behandle i én enkelt innsekvens. Dette refereres ofte til som modellens "oppmerksomhetsspenn" eller "kontekstvindu". Jo større kontekststørrelsen er, jo mer tekst kan modellen vurdere samtidig når den genererer et svar eller utfører en oppgave.

Forskjellige språkmodeller har varierende kontekststørrelser, fra noen hundre tokens til millioner av tokens. Som referanse kan et typisk avsnitt med tekst inneholde rundt 100-150 tokens, mens en hel bok kan inneholde titusenvis eller hundretusenvis av tokens.

> Det finnes til og med arbeid med effektive metoder for å skalere Transformer-baserte store språkmodeller (LLMs) til uendelig lange inputs med begrenset minne og beregning.

Hvorfor er kontekststørrelse viktig?

Kontekststørrelsen til en språkmodell har betydelig innvirkning på dens evne til å forstå og generere sammenhengende, kontekstuelt relevant tekst. Her er noen viktige grunner til hvorfor kontekststørrelse er viktig:

1. **Forståelse av langformat-innhold**: Modeller med større kontekststørrelse kan bedre forstå og analysere lengre tekster, som artikler, rapporter eller til og med hele bøker. Dette er avgjørende for oppgaver som dokumentsammendrag, spørsmålsbesvarelse og innholdsanalyse.

2. **Opprettholde sammenheng**: Et større kontekstvindu lar modellen opprettholde sammenheng og konsistens over lengre strekk av output. Dette er viktig for oppgaver som historiegenerering, dialogsystemer og innholdsproduksjon, hvor det er essensielt å opprettholde en konsistent fortelling eller tema. Det er også helt avgjørende når man bruker LLMer for å generere eller transformere strukturerte data.
3. **Fange opp langdistanseavhengigheter**: Noen språkoppgaver krever forståelse av forhold mellom ord eller fraser som er langt fra hverandre i en tekst. Modeller med større kontekststørrelse er bedre rustet til å fange opp disse langdistanseavhengighetene, som kan være viktige for oppgaver som stemningsanalyse, oversettelse og språkforståelse.
4. **Håndtere komplekse instruksjoner**: I anvendelser hvor språkmodeller brukes til å følge komplekse instruksjoner i flere trinn, tillater en større kontekststørrelse modellen å vurdere hele settet med instruksjoner når den genererer et svar, i stedet for bare de siste få ordene.

Eksempler på språkmodeller med forskjellige kontekststørrelser

Her er noen eksempler på språkmodeller med forskjellige kontekststørrelser:

- OpenAI GPT-3.5 Turbo: 4.095 tokens
- Mistral 7B Instruct: 32.768 tokens
- Anthropic Claude v1: 100.000 tokens
- OpenAI GPT-4 Turbo: 128.000 tokens
- Anthropic Claude v2: 200.000 tokens

- Google Gemini Pro 1.5: 2,8M tokens

Som du kan se, er det et bredt spekter av kontekststørrelser blant disse modellene, fra rundt 4.000 tokens for OpenAI GPT-3.5 Turbo-modellen til 200.000 tokens for Anthropic Claude v2-modellen. Noen modeller, som Googles PaLM 2 og OpenAIs GPT-4, tilbyr forskjellige varianter med større kontekststørrelser (f.eks. "32k"-versjoner), som kan håndtere enda lengre inputsekvenser. Og for øyeblikket (april 2024) skryter Google Gemini Pro av nesten 3 millioner tokens!

Det er verdt å merke seg at kontekststørrelsen kan variere avhengig av den spesifikke implementeringen og versjonen av en bestemt modell. For eksempel har den originale OpenAI GPT-4-modellen en kontekststørrelse på 8.191 tokens, mens de senere GPT-4-variantene som Turbo og 4o har en mye større kontekststørrelse på 128.000 tokens.

> Sam Altman har sammenlignet dagens kontekstbegrensninger med kilobytene av arbeidsminne som personlige dataprogrammerere måtte håndtere på 80-tallet, og sa at vi i nær fremtid vil kunne passe "alle dine personlige data" inn i konteksten til en stor språkmodell.

Velge riktig kontekststørrelse

Når man velger en språkmodell for en bestemt anvendelse, er det viktig å vurdere kontekststørrelseskravene for den aktuelle oppgaven. For oppgaver som involverer korte, isolerte tekstbiter, som stemningsanalyse eller enkel spørsmålsbesvarelse, kan en mindre kontekststørrelse være tilstrekkelig. For oppgaver som krever forståelse og generering av lengre, mer komplekse tekster, vil en større kontekststørrelse sannsynligvis være nødvendig.

Det er verdt å merke seg at større kontekststørrelser ofte kommer med økte beregningskostnader og tregere prosesseringstider, ettersom modellen må vurdere

mer informasjon når den genererer et svar. Derfor må du finne en balanse mellom kontekststørrelse og ytelse når du velger en språkmodell for din anvendelse.

> Hvorfor ikke bare velge modellen med størst kontekststørrelse og fylle den med så mye informasjon som mulig? Vel, bortsett fra ytelsesfaktorer er den andre hovedutfordringen kostnad. I mars 2024 vil en *enkelt* spørsmål-svar-syklus med Google Gemini Pro 1.5 med full kontekst koste deg nesten 8 dollar (USD). Hvis du har et brukstilfelle som rettferdiggjør den utgiften, all makt til deg! Men for de fleste anvendelser er det rett og slett for dyrt med flere størrelsesordener.

Å finne nåler i høystakker

Konseptet med å finne en nål i en høystakk har lenge vært en metafor for utfordringene med gjenfinning i store datasett. Innen store språkmodeller justerer vi denne analogien litt. Tenk deg at vi ikke bare leter etter én enkelt fakta gjemt i en omfattende tekst (som en fullstendig antologi av Paul Graham-essays), men flere fakta spredt utover. Dette scenariet ligner mer på å finne flere nåler i et vidstrakt jorde, ikke bare én høystakk. Her er poenget: vi må ikke bare finne disse nålene, men også veve dem sammen til en sammenhengende tråd.

Når store språkmodeller får i oppgave å gjenfinne og resonnere rundt flere fakta innebygd i lange kontekster, møter de en dobbel utfordring. Først er det det åpenbare problemet med gjenfinningsnøyaktighet – den synker naturlig når antall fakta øker. Dette er forventet; tross alt belaster det å holde styr på flere detaljer på tvers av en omfattende tekst selv de mest sofistikerte modellene.

For det andre, og kanskje mer kritisk, er utfordringen med å resonnere med disse faktaene. Det er én ting å plukke ut fakta; det er noe helt annet å syntetisere dem til en sammenhengende fortelling eller et svar. Det er her den virkelige testen kommer.

Ytelsen til store språkmodeller i resonneringsoppgaver har en tendens til å forringes mer enn i enkle gjenfinningsoppgaver. Denne forringelsen handler ikke bare om volum; det handler om det intrikate samspillet mellom kontekst, relevans og slutning.

Hvorfor skjer dette? Vel, tenk på dynamikken i hukommelse og oppmerksomhet i menneskelig kognisjon, som til en viss grad gjenspeiles i store språkmodeller. Når de behandler store mengder informasjon, kan store språkmodeller, i likhet med mennesker, miste oversikten over tidligere detaljer mens de tar inn nye. Dette er spesielt tilfelle i modeller som ikke er eksplisitt designet for å prioritere eller automatisk gå tilbake til tidligere tekstsegmenter.

Videre er språkmodellenes evne til å veve disse gjenfunne faktaene inn i et sammenhengende svar beslektet med narrativ konstruksjon. Dette krever ikke bare gjenfinning av informasjon, men en dyp forståelse og kontekstuell plassering, noe som fortsatt er en stor utfordring for dagens KI.

Så, hva betyr dette for oss som utviklere og integratorer av disse teknologiene? Vi må være svært bevisste på disse begrensningene når vi designer systemer som er avhengige av store språkmodeller for å håndtere komplekse, langformat-oppgaver. Å forstå at ytelsen kan forringes under visse forhold hjelper oss å sette realistiske forventninger og utvikle bedre reserveløsninger eller supplerende strategier.

Modaliteter: Utover tekst

Mens flertallet av språkmodeller i dag er fokusert på å behandle og generere tekst, er det en økende trend mot multimodale modeller som naturlig kan ta imot og produsere flere typer data, som bilder, lyd og video. Disse multimodale modellene åpner for nye muligheter for KI-drevne applikasjoner som kan forstå og generere innhold på tvers av forskjellige modaliteter.

Hva er modaliteter?

I sammenheng med språkmodeller refererer modaliteter til de forskjellige typene data som en modell kan behandle og generere. Den vanligste modaliteten er tekst, som inkluderer skriftlig språk i ulike former som bøker, artikler, nettsider og innlegg på sosiale medier. Det er imidlertid flere andre modaliteter som i økende grad blir innlemmet i språkmodeller:

- **Bilder**: Visuelle data som fotografier, illustrasjoner og diagrammer.
- **Lyd**: Lyddata som tale, musikk og miljølyder.
- **Video**: Bevegelige visuelle data, ofte ledsaget av lyd, som videoklipp og filmer.

Hver modalitet presenterer unike utfordringer og muligheter for språkmodeller. For eksempel krever bilder at modellen forstår visuelle konsepter og relasjoner, mens lyd krever at modellen behandler og genererer tale og andre lyder.

Multimodale språkmodeller

Multimodale språkmodeller er designet for å håndtere flere modaliteter innenfor én enkelt modell. Disse modellene har vanligvis spesialiserte komponenter eller lag som både kan forstå inndata og generere utdata i forskjellige modaliteter. Noen bemerkelsesverdige eksempler på multimodale språkmodeller inkluderer:

- **OpenAIs GPT-4o**: GPT-4o er en stor språkmodell som naturlig forstår og behandler talelyd i tillegg til tekst. Denne egenskapen gjør at GPT-4o kan utføre oppgaver som å transkribere talespråk, generere tekst fra lydinput og gi svar basert på muntlige spørsmål.
- **OpenAIs GPT-4 med visuell input**: GPT-4 er en stor språkmodell som kan behandle både tekst og bilder. Når den får et bilde som input, kan GPT-4 analysere innholdet i bildet og generere tekst som beskriver eller responderer på den visuelle informasjonen.

- **Googles Gemini**: Gemini er en multimodal modell som kan håndtere tekst, bilder og video. Den bruker en enhetlig arkitektur som muliggjør kryssmodal forståelse og generering, og muliggjør oppgaver som bildeteksting, videooppsummering og visuell spørsmålsbesvarelse.

- **DALL-E og Stable Diffusion**: Selv om disse ikke er språkmodeller i tradisjonell forstand, demonstrerer disse modellene kraften i multimodal AI ved å generere bilder fra tekstbeskrivelser. De viser potensialet for modeller som kan oversette mellom ulike modaliteter.

Fordeler og bruksområder for multimodale modeller

Multimodale språkmodeller tilbyr flere fordeler og muliggjør et bredt spekter av bruksområder, inkludert:

- **Forbedret forståelse**: Ved å behandle informasjon fra flere modaliteter kan disse modellene få en mer omfattende forståelse av verden, lignende måten mennesker lærer fra ulike sensoriske inndata.
- **Kryssmodal generering**: Multimodale modeller kan generere innhold i én modalitet basert på inndata fra en annen, som å lage et bilde fra en tekstbeskrivelse eller generere et videosammendrag fra en skriftlig artikkel.
- **Tilgjengelighet**: Multimodale modeller kan gjøre informasjon mer tilgjengelig ved å oversette mellom modaliteter, som å generere tekstbeskrivelser av bilder for synshemmede brukere eller lage lydversjoner av skriftlig innhold.
- **Kreative anvendelser**: Multimodale modeller kan brukes til kreative oppgaver som å generere kunst, musikk eller videoer basert på tekstlige prompts, noe som åpner nye muligheter for kunstnere og innholdsskapere.

Ettersom multimodale språkmodeller fortsetter å utvikle seg, vil de sannsynligvis spille en stadig viktigere rolle i utviklingen av AI-drevne applikasjoner som kan forstå og

generere innhold på tvers av flere modaliteter. Dette vil muliggjøre mer naturlig og intuitiv interaksjon mellom mennesker og AI-systemer, samt åpne for nye muligheter innen kreativ uttrykk og kunnskapsformidling.

Leverandørøkosystemer

Når det gjelder å inkorporere store språkmodeller (LLMs) i applikasjoner, har du et voksende utvalg av alternativer å velge mellom. Hver større LLM-leverandør, som OpenAI, Anthropic, Google og Cohere, tilbyr sitt eget økosystem av modeller, API-er og verktøy. Å velge riktig leverandør innebærer å vurdere ulike faktorer, inkludert prising, ytelse, innholdsfiltrering, datapersonvern og tilpasningsmuligheter.

OpenAI

OpenAI er en av de mest kjente leverandørene av LLMs, med sin GPT-serie (GPT-3, GPT-4) som er mye brukt i ulike applikasjoner. OpenAI tilbyr et brukervennlig API som lar deg enkelt integrere modellene deres i applikasjoner. De tilbyr en rekke modeller med ulike kapabiliteter og prisnivåer, fra innstegsmodellen Ada til den kraftige Davinci-modellen.

OpenAIs økosystem inkluderer også verktøy som OpenAI Playground, som lar deg eksperimentere med prompts og finjustere modeller for spesifikke brukstilfeller. De tilbyr innholdsfiltreringsalternativer for å hindre generering av upassende eller skadelig innhold.

Når jeg bruker OpenAIs modeller direkte, stoler jeg på Alex Rudalls ruby-openai-bibliotek.

Anthropic

Anthropic er en annen stor aktør innen LLM-området, der deres Claude-modeller blir stadig mer populære for sin sterke ytelse og etiske hensyn. Anthropic fokuserer på

å utvikle trygge og ansvarlige AI-systemer, med stor vekt på innholdsfiltrering og unngåelse av skadelige outputs.

Anthropics økosystem inkluderer Claude API, som lar deg integrere modellen i applikasjonene deres, samt verktøy for prompteknikk og finjustering. De tilbyr også Claude Instant-modellen, som inkorporerer websøk-funksjoner for mer oppdaterte og faktabaserte svar.

Når jeg bruker Anthropics modeller direkte, stoler jeg på Alex Rudalls anthrophic-bibliotek.

Google

Google har utviklet flere kraftige LLMs, inkludert Gemini, BERT, T5 og PaLM. Disse modellene er kjent for sin sterke ytelse på et bredt spekter av naturlig språkbehandlingsoppgaver. Googles økosystem inkluderer TensorFlow- og Keras-bibliotekene, som tilbyr verktøy og rammeverk for å bygge og trene maskinlæringsmodeller.

Google tilbyr også en Cloud AI Platform, som lar deg enkelt distribuere og skalere modellene deres i skyen. De tilbyr en rekke forhåndstrente modeller og API-er for oppgaver som stemningsanalyse, entitetsgjenkjenning og oversettelse.

Meta

Meta, tidligere kjent som Facebook, er dypt involvert i utviklingen av store språkmodeller, fremhevet av deres utgivelse av modeller som LLaMA og OPT. Disse modellene utmerker seg med sin sterke ytelse i ulike språkoppgaver og er i stor grad tilgjengelige gjennom åpen kildekode-kanaler, som støtter Metas engasjement for forskning og fellesskapssamarbeid.

Metas økosystem er primært bygget rundt PyTorch, et åpen kildekode-maskinlæringsbibliotek som foretrekkes for sine dynamiske beregningsevner og fleksibilitet, som tilrettelegger for innovativ AI-forskning og -utvikling.

I tillegg til deres tekniske tilbud legger Meta stor vekt på etisk AI-utvikling. De implementerer robust innholdsfiltrering og fokuserer på å redusere skjevheter, i tråd med deres overordnede mål om sikkerhet og ansvarlighet i AI-applikasjoner.

Cohere

Cohere er en nyere aktør innen LLM-området, som fokuserer på å gjøre LLM-er mer tilgjengelige og enklere å bruke enn konkurrentene. Deres økosystem inkluderer Cohere API, som gir tilgang til en rekke forhåndstrente modeller for oppgaver som tekstgenerering, klassifisering og oppsummering.

Cohere tilbyr også verktøy for prompt-utvikling, finjustering og innholdsfiltrering. De legger vekt på databeskyttelse og sikkerhet, med funksjoner som kryptert datalagring og tilgangskontroll.

Ollama

Ollama er en selvdriftet plattform som lar brukere administrere og distribuere ulike store språkmodeller (LLM-er) lokalt på sine maskiner, noe som gir dem fullstendig kontroll over AI-modellene uten å være avhengig av eksterne skytjenester. Denne løsningen er ideell for de som prioriterer personvern og ønsker å håndtere sine AI-operasjoner internt.

Plattformen støtter en rekke modeller, inkludert versjoner av Llama, Phi, Gemma og Mistral, som varierer i størrelse og beregningskrav. Ollama gjør det enkelt å laste ned og kjøre disse modellene direkte fra kommandolinjen ved hjelp av enkle kommandoer som `ollama run <model_name>`, og den er designet for å fungere på tvers av ulike operativsystemer inkludert macOS, Linux og Windows.

For utviklere som ønsker å integrere åpen kildekode-modeller i applikasjonene sine uten å bruke en ekstern API, tilbyr Ollama et CLI for å administrere modellenes livssyklus på lignende måte som containeradministrasjonsverktøy. Den støtter også egendefinerte konfigurasjoner og prompts, noe som muliggjør en høy grad av tilpasning for å skreddersy modellene til spesifikke behov eller bruksområder.

Ollama er spesielt egnet for teknisk kyndige brukere og utviklere på grunn av sitt kommandolinjegrensesnitt og fleksibiliteten den tilbyr i administrasjon og distribusjon av AI-modeller. Dette gjør det til et kraftig verktøy for bedrifter og enkeltpersoner som krever robuste AI-kapabiliteter uten å gå på kompromiss med sikkerhet og kontroll.

Multi-modell plattformer

I tillegg finnes det leverandører som er vert for et bredt utvalg av åpen kildekode-modeller, som Together.ai og Groq. Disse plattformene tilbyr fleksibilitet og tilpasningsmuligheter, som lar deg kjøre og i noen tilfeller til og med finjustere åpen kildekode-modeller i henhold til dine spesifikke behov. For eksempel gir Together.ai tilgang til en rekke åpen kildekode-LLM-er, som gjør det mulig for brukere å eksperimentere med forskjellige modeller og konfigurasjoner. Groq fokuserer på å levere ultrahøy ytelse som på tidspunktet for denne bokens utgivelse virker nesten magisk

Velge en LLM-leverandør

Når du velger en LLM-leverandør, bør du vurdere faktorer som:

- **Prising**: Forskjellige leverandører tilbyr ulike prismodeller, fra betal-per-bruk til abonnementsbaserte planer. Det er viktig å vurdere forventet bruk og budsjett når man velger en leverandør.
- **Ytelse**: Ytelsen til LLM-er kan variere betydelig mellom leverandører, så det er viktig å teste modeller på spesifikke brukstilfeller før man tar en beslutning.
- **Innholdsfiltrering**: Avhengig av applikasjonen kan innholdsfiltrering være en kritisk faktor. Noen leverandører tilbyr mer robuste innholdsfilteringsalternativer enn andre.
- **Personvern**: Hvis applikasjonen håndterer sensitive brukerdata, er det viktig å velge en leverandør med sterke personverns- og sikkerhetspraksis.

- **Tilpasning**: Noen leverandører tilbyr mer fleksibilitet når det gjelder finjustering og tilpasning av modeller for spesifikke brukstilfeller.

Til syvende og sist avhenger valget av LLM-leverandør av de spesifikke kravene og begrensningene til applikasjonen. Ved å nøye evaluere alternativene og vurdere faktorer som prising, ytelse og personvern, kan du velge den leverandøren som best møter dine behov.

Det er også verdt å merke seg at LLM-landskapet er i konstant utvikling, med nye leverandører og modeller som dukker opp regelmessig. Du bør holde deg oppdatert på de nyeste utviklingene og være åpen for å utforske nye alternativer etter hvert som de blir tilgjengelige.

OpenRouter

Gjennom denne boken vil jeg utelukkende bruke OpenRouter som min foretrukne API-leverandør. Grunnen er enkel: det er en alt-i-ett-løsning for alle de mest populære kommersielle og åpen kildekode-modellene. Hvis du er ivrig etter å komme i gang med AI-koding, er et av de beste stedene å starte med mitt eget OpenRouter Ruby-bibliotek.

Tenke på ytelse

Når man integrerer språkmodeller i applikasjoner, er ytelse en kritisk faktor å ta hensyn til. Ytelsen til en språkmodell kan måles i form av dens *latens* (tiden det tar å generere et svar) og *gjennomstrømning* (antall forespørsler den kan håndtere per tidsenhet).

Tid til første token (TTFT) er enda en essensiell ytelsesmetrikk, særlig relevant for chatbots og applikasjoner som krever interaktive sanntidssvar. TTFT måler latensen fra øyeblikket en brukers forespørsel mottas til øyeblikket det første ordet (eller tokenet) i svaret genereres. Denne metrikken er avgjørende for å opprettholde en sømløs og

engasjerende brukeropplevelse, ettersom forsinkede svar kan føre til brukerfrustrasjon og manglende engasjement.

Disse ytelsesmetrikker kan ha betydelig innvirkning på brukeropplevelsen og applikasjonens skalerbarhet.

Flere faktorer kan påvirke ytelsen til en språkmodell, inkludert:

Parameterantall: Større modeller med flere parametere krever generelt mer dataressurser og kan ha høyere latens og lavere gjennomstrømning sammenlignet med mindre modeller.

Maskinvare: Ytelsen til en språkmodell kan variere betydelig basert på maskinvaren den kjører på. Skyleverandører tilbyr GPU- og TPU-instanser optimalisert for maskinlæringsarbeidsbelastninger, som kan gi betydelig akselerasjon av modellinferens.

En av de fine tingene med OpenRouter er at for mange av modellene de tilbyr, får du valget mellom skyleverandører med forskjellige ytelsesprofiler og kostnader.

Kvantisering: Kvantiseringsteknikker kan brukes for å redusere minneforbruket og beregningskravene til en modell ved å representere vekter og aktiveringer med datatyper av lavere presisjon. Dette kan forbedre ytelsen uten å ofre kvaliteten betydelig. Som applikasjonsutvikler vil du sannsynligvis ikke være involvert i å trene dine egne modeller på forskjellige kvantiseringsnivåer, men det er greit å være kjent med terminologien.

Gruppering: Behandling av flere forespørsler samtidig i grupper kan forbedre gjennomstrømningen ved å fordele overhead for modellinnlasting og dataoverføring.

Mellomlagring: Mellomlagring av resultater fra hyppig brukte prompts eller innsekvenser kan redusere antall inferensforespørsler og forbedre den generelle ytelsen.

Når man velger en språkmodell for en produksjonsapplikasjon, er det viktig å måle ytelsen på representative arbeidsbelastninger og maskinvarekonfigurasjoner. Dette kan

hjelpe med å identifisere potensielle flaskehalser og sikre at modellen kan møte de nødvendige ytelsesmålene.

Det er også verdt å vurdere avveiningene mellom modellytelse og andre faktorer som kostnad, fleksibilitet og integreringsevne. For eksempel kan bruk av en mindre, rimeligere modell med lavere latens være å foretrekke for applikasjoner som krever sanntidssvar, mens en større, kraftigere modell kan være bedre egnet for gruppeprosessering eller komplekse resonneringsoppgaver.

Eksperimentere med forskjellige LLM-modeller

Valg av LLM er sjelden en permanent beslutning. Ettersom nye og forbedrede modeller lanseres regelmessig, er det lurt å bygge applikasjoner på en modulær måte som tillater utskifting av forskjellige språkmodeller over tid. Prompts og datasett kan ofte gjenbrukes på tvers av modeller med minimale endringer. Dette gjør det mulig å dra nytte av de nyeste fremskrittene innen språkmodellering uten å måtte redesigne applikasjonene fullstendig.

Muligheten til å enkelt bytte mellom et bredt utvalg av modeller er enda en grunn til at jeg elsker OpenRouter.

Når man oppgraderer til en ny språkmodell, er det viktig å grundig teste og validere dens ytelse og outputkvalitet for å sikre at den oppfyller applikasjonens krav. Dette kan innebære å trene på nytt eller finjustere modellen på domenespesifikke data, samt oppdatere eventuelle nedstrømskomponenter som er avhengige av modellens output.

Ved å designe applikasjoner med ytelse og modularitet i tankene, kan du skape skalerbare, effektive og fremtidssikre systemer som kan tilpasse seg det raskt utviklende landskapet av språkmodelleringsteknologi.

Sammensatte AI-systemer

Før vi avslutter vår introduksjon, er det verdt å nevne at før 2023 og eksplosjonen av interesse for generativ AI utløst av ChatGPT, var tradisjonelle AI-tilnærminger vanligvis avhengige av integrasjon av enkeltstående, lukkede modeller. I motsetning til dette utnytter *sammensatte AI-systemer* komplekse rørledninger av sammenkoblede komponenter som jobber sammen for å oppnå intelligent oppførsel.

I kjernen består sammensatte AI-systemer av flere moduler, hver designet for å utføre spesifikke oppgaver eller funksjoner. Disse modulene kan inkludere generatorer, innhentere, rangerere, klassifikatorer og forskjellige andre spesialiserte komponenter. Ved å bryte ned det overordnede systemet i mindre, fokuserte enheter, kan utviklere skape mer fleksible, skalerbare og vedlikeholdbare AI-arkitekturer.

En av de viktigste fordelene med sammensatte AI-systemer er deres evne til å kombinere styrkene fra forskjellige AI-teknikker og modeller. For eksempel kan et system bruke en stor språkmodell (LLM) for naturlig språkforståelse og generering, mens det bruker en separat modell for informasjonsgjenfinning eller regelbasert beslutningstaking. Denne modulære tilnærmingen lar deg velge de beste verktøyene og teknikkene for hver spesifikke oppgave, i stedet for å stole på en universalløsning.

Imidlertid byr bygging av sammensatte AI-systemer også på unike utfordringer. Spesielt krever det å sikre systemets generelle sammenheng og konsistens robuste mekanismer for testing, overvåking og styring.

Fremveksten av kraftige LLM-er som GPT-4 lar oss eksperimentere med sammensatte AI-systemer enklere enn noensinne før, fordi disse avanserte modellene er i stand til å håndtere flere roller innenfor et sammensatt system, som klassifisering, rangering og generering, i tillegg til deres naturlige språkforståelsesevner. Denne allsidigheten gjør det mulig for utviklere å raskt prototype og iterere på sammensatte AI-arkitekturer, noe som åpner nye muligheter for utvikling av intelligente applikasjoner.

Distribusjonsmønstre for sammensatte AI-systemer

Sammensatte AI-systemer kan distribueres ved hjelp av ulike mønstre, som hver er designet for å håndtere spesifikke krav og brukstilfeller. La oss utforske fire vanlige distribusjonsmønstre: Spørsmål og Svar, Flerarent/Agentiske Problemløsere, Konversasjons-AI, og CoPiloter.

Spørsmål og Svar

Spørsmål og svar (Q&A)-systemer fokuserer på å levere informasjonsgjenfinning som er forbedret med forståelsesevnene til AI-modeller for å fungere som mer enn bare en søkemotor. Ved å kombinere kraftige språkmodeller med eksterne kunnskapskilder ved hjelp av Gjenfinningsforsterket Generering (RAG), unngår spørsmål og svar-systemer hallusinasjoner og gir nøyaktige og kontekstuelt relevante svar på brukerforespørsler.

Hovedkomponentene i et LLM-basert Q&A-system inkluderer:

- **Spørsmålsforståelse og omformulering**: Analysering av brukerforespørsler og omformulering av disse for bedre å matche de underliggende kunnskapskildene.
- **Kunnskapsgjenfinning**: Henting av relevant informasjon fra strukturerte eller ustrukturerte datakilder basert på den omformulerte forespørselen.
- **Svargenerering**: Generering av sammenhengende og informative svar ved å integrere den gjenfunne kunnskapen med språkmodellens generative evner.

RAG-delsystemer er spesielt viktige i Q&A-domener hvor det er avgjørende å gi nøyaktig og oppdatert informasjon, som kundesupport, kunnskapshåndtering, eller utdanningsapplikasjoner

Flerarent/Agentiske Problemløsere

Flerarent-systemer, også kjent som *agentiske* systemer, består av flere autonome agenter som samarbeider for å løse komplekse problemer. Hver agent har en spesifikk rolle,

et sett med ferdigheter og tilgang til relevante verktøy eller informasjonskilder. Ved å samarbeide og utveksle informasjon kan disse agentene takle oppgaver som ville vært vanskelige eller umulige for en enkelt agent å håndtere alene.

Hovedprinsippene for flerarent-problemløsere inkluderer:

- **Spesialisering**: Hver agent fokuserer på et spesifikt aspekt av problemet, ved å utnytte sine unike evner og kunnskap.
- **Samarbeid**: Agenter kommuniserer og koordinerer sine handlinger for å oppnå et felles mål, ofte gjennom meldingsutveksling eller delt minne.
- **Tilpasningsevne**: Systemet kan tilpasse seg endrede forhold eller krav ved å justere rollene og atferden til individuelle agenter.

Flerarent-systemer er godt egnet for applikasjoner som krever distribuert problemløsning, som forsyningskjedeoptimalisering, trafikkstrying, eller beredskapsplanlegging

Konversasjons-AI

Konversasjons-AI-systemer muliggjør naturlig språkinteraksjon mellom brukere og intelligente agenter. Disse systemene kombinerer naturlig språkforståelse, dialoghåndtering og språkgenerering for å gi engasjerende og personlige samtaleopplevelser.

Hovedkomponentene i et konversasjons-AI-system inkluderer:

- **Intensjonsgjenkjenning**: Identifisering av brukerens intensjon basert på deres inndata, som å stille et spørsmål, komme med en forespørsel eller uttrykke en følelse.
- **Entitetsuttrekking**: Uttrekking av relevante enheter eller parametere fra brukerens inndata, som datoer, steder eller produktnavn.

- **Dialoghåndtering**: Vedlikehold av samtalens tilstand, bestemmelse av passende svar basert på brukerens intensjon og kontekst, og håndtering av flertursamtaler.
- **Svargenerering**: Generering av menneskelige svar ved hjelp av språkmodeller, maler eller gjenfinningsbaserte metoder.

Konversasjons-AI-systemer brukes ofte i kundeservice-chatboter, virtuelle assistenter, og stemmestyrte grensesnitt. Som nevnt tidligere er de fleste tilnærmingene, mønstrene og kodeeksemplene i denne boken direkte hentet fra mitt arbeid med et stort konversasjons-AI-system kalt Olympia

CoPilots

CoPilots er AI-drevne assistenter som jobber sammen med menneskelige brukere for å forbedre deres produktivitet og beslutningsevne. Disse systemene utnytter en kombinasjon av naturlig språkbehandling, maskinlæring og domenespesifikk kunnskap for å gi intelligente anbefalinger, automatisere oppgaver og tilby kontekstuell støtte.

Hovedfunksjoner i CoPilots inkluderer:

- **Personalisering**: Tilpasning til individuelle brukerpreferanser, arbeidsflyter og kommunikasjonsstiler.
- **Proaktiv assistanse**: Forutser brukerbehov og tilbyr relevante forslag eller handlinger uten eksplisitte forespørsler.
- **Kontinuerlig læring**: Forbedrer ytelsen over tid ved å lære fra brukerrespons, interaksjoner og data.

CoPilots blir i økende grad brukt i ulike domener, som programvareutvikling (f.eks. kodekomplettering og feildeteksjon), kreativ skriving (f.eks. innholdsforslag og redigering), og dataanalyse (f.eks. innsikt og visualiseringsanbefalinger)

Disse implementeringsmønstrene viser allsidigheten og potensialet til sammensatte AI-systemer. Ved å forstå egenskapene og bruksområdene for hvert mønster, kan du ta

informerte beslutninger når du designer og implementerer intelligente applikasjoner. Selv om denne boken ikke spesifikt handler om implementering av sammensatte AI-systemer, gjelder mange, om ikke alle, av de samme tilnærmingene og mønstrene for integrering av diskrete AI-komponenter innen ellers tradisjonell applikasjonsutvikling.

Roller i sammensatte AI-systemer

Sammensatte AI-systemer er bygget på et fundament av sammenkoblede moduler, der hver modul er designet for å utføre en spesifikk rolle. Disse modulene samarbeider for å skape intelligent oppførsel og løse komplekse problemer. Det er nyttig å være kjent med disse rollene når man tenker på hvor man kan implementere eller erstatte deler av applikasjonen med diskrete AI-komponenter.

Generator

Generatorer er ansvarlige for å produsere nye data eller innhold basert på lærte mønstre eller input-prompts. AI-verdenen har mange forskjellige typer generatorer, men i konteksten av språkmodellene som presenteres i denne boken, kan generatorer skape menneskelignende tekst, fullføre uferdige setninger eller generere svar på brukerforespørsler. De spiller en avgjørende rolle i oppgaver som innholdsproduksjon, dialoggenerering og dataforsterkning.

Innhenter

Innhentere brukes til å søke og hente relevant informasjon fra store datasett eller kunnskapsbaser. De benytter teknikker som semantisk søk, nøkkelordmatchning eller vektorlikhet for å finne de mest relevante datapunktene basert på en gitt forespørsel eller kontekst. Innhentere er essensielle for oppgaver som krever rask tilgang til spesifikk informasjon, som spørsmålsbesvarelse, faktasjekking eller innholdsanbefaling.

Rangerer

Rangerere er ansvarlige for å ordne eller prioritere et sett med elementer basert på bestemte kriterier eller relevanspoeng. De tildeler vekter eller poeng til hvert element og sorterer dem deretter. Rangerere brukes ofte i søkemotorer, anbefalingssystemer eller enhver applikasjon hvor det er viktig å presentere de mest relevante resultatene for brukerne.

Klassifikator

Klassifikatorer brukes til å kategorisere eller merke datapunkter basert på forhåndsdefinerte klasser eller kategorier. De lærer fra merket treningsdata og forutsier deretter klassen til nye, usette tilfeller. Klassifikatorer er grunnleggende for oppgaver som stemningsanalyse, spam-deteksjon eller bildegjenkjenning, hvor målet er å tilordne en spesifikk kategori til hver input.

Verktøy og agenter

I tillegg til disse kjernefunksjonene inkorporerer sammensatte AI-systemer ofte verktøy og agenter for å forbedre sin funksjonalitet og tilpasningsevne:

- **Verktøy**: Verktøy er diskrete programvarekomponenter eller API-er som utfører spesifikke handlinger eller beregninger. De kan påkalles av andre moduler, som generatorer eller innhentere, for å utføre deloppgaver eller samle tilleggsinformasjon. Eksempler på verktøy inkluderer nettsøkemotorer, kalkulatorer eller datavisualiseringsbiblioteker.
- **Agenter**: Agenter er autonome enheter som kan oppfatte sitt miljø, ta beslutninger og utføre handlinger for å oppnå spesifikke mål. De er ofte avhengige av en kombinasjon av forskjellige AI-teknikker, som planlegging, resonnering og læring, for å operere effektivt under dynamiske eller usikre forhold. Agenter kan brukes til å modellere kompleks oppførsel eller koordinere handlingene til flere moduler innen et sammensatt AI-system.

I et rent sammensatt AI-system blir interaksjonen mellom disse komponentene orkestrert gjennom veldefinerte grensesnitt og kommunikasjonsprotokoller. Data flyter mellom moduler, der outputen fra én komponent tjener som input for en annen. Denne modulære arkitekturen muliggjør fleksibilitet, skalerbarhet og vedlikeholdbarhet, ettersom individuelle komponenter kan oppdateres, erstattes eller utvides uten å påvirke hele systemet.

Ved å utnytte kraften i disse komponentene og deres interaksjoner, kan sammensatte AI-systemer takle komplekse, virkelige problemer som krever en kombinasjon av forskjellige AI-kapabiliteter. Når vi utforsker tilnærmingene og mønstrene for integrering av AI i applikasjonsutvikling, husk at de samme prinsippene og teknikkene som brukes i sammensatte AI-systemer kan anvendes for å skape intelligente, adaptive og brukersentrerte applikasjoner.

I de følgende kapitlene i Del 1 vil vi dykke dypere inn i de grunnleggende tilnærmingene og teknikkene for integrering av AI-komponenter i din applikasjonsutviklingsprosess. Fra prompt-teknikk og innhentingsforsterket generering til selvhelbredende data og intelligent arbeidsflytorkestrering, vil vi dekke et bredt spekter av mønstre og beste praksis for å hjelpe deg med å bygge banebrytende AI-drevne applikasjoner.

Del 1: Grunnleggende tilnærminger og teknikker

Denne delen av boken presenterer forskjellige måter å integrere bruken av AI i applikasjonene dine. Kapitlene dekker en rekke beslektede tilnærminger og teknikker, fra mer overordnede konsepter som Narrow The Path og Retrieval Augmented Generation, helt ned til ideer for programmering av ditt eget abstraksjonslag på toppen av LLM chatteferdigstillings-APIer.

Målet med denne delen av boken er å hjelpe deg med å forstå hvilke typer atferd du kan implementere med AI, før vi går for dypt inn i spesifikke implementeringsmønstre som er fokuset i Del 2.

Tilnærmingene i Del 1 er basert på ideer jeg har brukt i min kode, klassiske mønstre for enterpriseapplikasjonsarkitektur og integrasjon, samt metaforer jeg har brukt når jeg har forklart AI-mulighetene til andre mennesker, inkludert ikke-tekniske forretningsinteressenter.

Innsnevre stien

“Innsnevre stien” handler om å fokusere KI-en på den aktuelle oppgaven. Jeg bruker det som et mantra når jeg blir frustrert over at KI-en oppfører seg “dum” eller på uventede måter. Mantraet minner meg om at feilen sannsynligvis er min egen, og at jeg antagelig burde innsnevre stien enda mer.

Behovet for å innsnevre stien oppstår fra den enorme mengden kunnskap som finnes i store språkmodeller, spesielt verdensklassemodeller som de fra OpenAI og Anthropic som bokstavelig talt har billioner av parametere.

Å ha tilgang til et så bredt kunnskapsspekter er utvilsomt kraftfullt og produserer emergent atferd som sinnsteori og evnen til å resonnere på menneskelige måter. Denne banebrytende informasjonsmengden skaper imidlertid utfordringer når det gjelder å generere presise og nøyaktige svar på spesifikke prompts, spesielt hvis disse promptene er ment å utvise deterministisk atferd som kan integreres med "normal" programvareutvikling og algoritmer.

Flere faktorer fører til utfordringene.

Informasjonsoverbelastning: Store språkmodeller er trent på massive mengder data som spenner over ulike domener, kilder og tidsperioder. Denne omfattende kunnskapen gjør dem i stand til å engasjere seg i diverse emner og generere svar basert på en bred forståelse av verden. Når modellen står overfor en spesifikk prompt, kan den imidlertid slite med å filtrere ut irrelevant, motstridende eller utdatert/foreldet informasjon, noe som fører til svar som mangler fokus eller nøyaktighet. Avhengig av hva du prøver å gjøre, kan den enorme mengden *motstridende* informasjon som er tilgjengelig for modellen lett overvelde dens evne til å gi svaret eller atferden du søker.

Kontekstuell tvetydighet: Gitt det enorme *latente rommet* av kunnskap, kan store språkmodeller støte på tvetydighet når de prøver å forstå *konteksten* i prompten din. Uten riktig innsnevring eller veiledning kan modellen generere svar som er tangentielt relaterte, men ikke direkte relevante for dine intensjoner. Denne typen feil fører til svar som er utenfor tema, inkonsistente eller ikke møter dine uttalte behov. I dette tilfellet refererer innsnevring av stien til kontekst *disambiguering*, som sikrer at konteksten du gir får modellen til å fokusere kun på den mest relevante informasjonen i sin grunnleggende kunnskap.

Merk: Når du begynner med "prompt-konstruksjon" er det mye mer sannsynlig at du ber modellen gjøre ting uten å forklare det ønskede resultatet ordentlig; det krever øvelse å ikke være tvetydig!

Temporære inkonsistenser: Siden språkmodeller er trent på data som ble opprettet

på forskjellige tidspunkter, kan de besitte kunnskap som er utdatert, erstattet eller ikke lenger nøyaktig. For eksempel kan informasjon om aktuelle hendelser, vitenskapelige oppdagelser eller teknologiske fremskritt ha utviklet seg siden modellens treningsdata ble samlet inn. Uten å innsnevre stien for å prioritere nyere og mer pålitelige kilder, kan modellen generere svar basert på utdatert eller feil informasjon, noe som fører til unøyaktigheter og inkonsistenser i utdataene.

Domenespesifikke nyanser: Forskjellige domener og felt har sin egen spesifikke terminologi, konvensjoner og kunnskapsbaser. Tenk på praktisk talt hvilken som helst TLA (trebokstavsforkortelse) og du vil innse at de fleste av dem har mer enn én betydning. For eksempel kan MSK referere til Amazon's Managed Streaming for Apache Kafka, Memorial Sloan Kettering Cancer Center, eller det menneskelige muskel- og skjelettsystemet.

Når en prompt krever ekspertise innen et bestemt domene, kan en stor språkmodells generiske kunnskap være utilstrekkelig for å gi nøyaktige og nyanserte svar. Å innsnevre stien ved å fokusere på domenespesifikk informasjon, enten gjennom prompt-konstruksjon eller gjenfinningsforsterket generering, lar modellen generere svar som er mer på linje med ditt spesifikke domenes krav og forventninger.

Latent rom: Ubegripelig stort

Når jeg nevner det "latente rommet" i en språkmodell, refererer jeg til det enorme, flerdimensjonale landskapet av kunnskap og informasjon som modellen har lært under treningsprosessen. Det er som et skjult rike inni modellens nevrale nettverk, hvor alle mønstre, assosiasjoner og representasjoner av språk er lagret.

Forestill deg at du utforsker et stort, ukartlagt territorium fylt med utallige sammenkoblede noder. Hver node representerer en informasjonsbit, et konsept eller en relasjon som modellen har lært. Når du navigerer gjennom dette rommet, vil du oppdage at noen noder er nærmere hverandre, noe som indikerer en sterk forbindelse

eller likhet, mens andre er lengre fra hverandre, noe som antyder en svakere eller mer fjern relasjon.

Utfordringen med latent rom er at det er utrolig komplekst og høydimensjonalt. Tenk på det som like enormt som vårt fysiske univers, med sine galaksehoper og enorme, ufattelige avstander med tomt rom mellom dem.

Fordi det inneholder tusenvis av dimensjoner, er det latente rommet ikke direkte observerbart eller tolkbart for mennesker. Det er en abstrakt representasjon som modellen bruker internt for å behandle og generere språk. Når du gir modellen en innledende prompt, kartlegger den i hovedsak denne prompten til en bestemt plassering i det latente rommet. Modellen bruker deretter den omkringliggende informasjonen og forbindelsene i dette rommet for å generere et svar.

Saken er at modellen har lært en enorm mengde informasjon fra treningsdataene sine, og ikke alt er relevant eller nøyaktig for en gitt oppgave. Det er derfor innsnevring av stien blir så viktig. Ved å gi klare instruksjoner, eksempler og kontekst i dine prompts, leder du i hovedsak modellen til å fokusere på spesifikke regioner innenfor det latente rommet som er mest relevante for ønsket resultat.

En annen måte å tenke på det er som å bruke en spotlight i et helt mørkt museum. Hvis du noen gang har besøkt Louvre eller Metropolitan Museum of Art, så er det den type skala jeg snakker om. Det latente rommet er museet, fylt med utallige objekter og detaljer. Din prompt er spotlighten som lyser opp spesifikke områder og trekker modellens oppmerksomhet mot den viktigste informasjonen. Uten denne veiledningen kan modellen vandre målløst gjennom det latente rommet og plukke opp irrelevant eller motstridende informasjon underveis.

Når du jobber med språkmodeller og utformer dine prompts, husk konseptet med latent rom. Målet ditt er å navigere effektivt i dette enorme kunnskapslandskapet, og styre modellen mot den mest relevante og nøyaktige informasjonen for din oppgave. Ved å innsnevre stien og gi klar veiledning kan du låse opp det fulle potensialet i modellens latente rom og generere høykvalitets, sammenhengende svar.

Mens de tidligere beskrivelsene av språkmodeller og det latente rommet de navigerer i kan virke litt magisk eller abstrakt, er det viktig å forstå at prompts ikke er trylleformler eller besvergelser. Måten språkmodeller fungerer på er forankret i prinsippene fra lineær algebra og sannsynlighetsteori.

I kjernen er språkmodeller probabilistiske modeller av tekst, på samme måte som en normalfordelingskurve er en statistisk modell av data. De trenes gjennom en prosess kalt autoregressiv modellering, hvor modellen lærer å forutsi sannsynligheten for det neste ordet i en sekvens basert på ordene som kommer før det. Under trening starter modellen med tilfeldige vekter og justerer dem gradvis for å tilordne høyere sannsynligheter til tekst som ligner på virkelige eksempler den ble trent på.

Imidlertid gir det ikke den beste intuisjonen å tenke på språkmodeller som enkle statistiske modeller, som lineær regresjon. En mer passende analogi er å tenke på dem som probabilistiske programmer, som er modeller som tillater manipulering av tilfeldige variabler og kan representere komplekse statistiske relasjoner.

Probabilistiske programmer kan representeres av grafiske modeller, som gir en visuell måte å forstå avhengighetene og relasjonene mellom variabler i modellen. Dette perspektivet kan gi verdifull innsikt i hvordan komplekse tekstgenereringsmodeller som GPT-4 og Claude fungerer.

I artikkelen "Language Model Cascades" av Dohan et al., går forfatterne i dybden på hvordan probabilistiske programmer kan anvendes på språkmodeller. De viser hvordan dette rammeverket kan brukes til å forstå oppførselen til disse modellene og guide utviklingen av mer effektive promptingstrategier.

En viktig innsikt fra dette probabilistiske perspektivet er at språkmodellen i hovedsak skaper en portal til et alternativt univers hvor de ønskede dokumentene eksisterer. Modellen tilordner vekter til alle mulige dokumenter basert på deres sannsynlighet, og innsnevrer effektivt rommet av muligheter for å fokusere på de mest relevante.

Dette bringer oss tilbake til hovedtemaet om "innsnevring av stien." Hovedmålet med prompting er å betinge den probabilistiske modellen på en måte som fokuserer massen av

dens prediksjoner, og spisser seg inn mot den spesifikke informasjonen eller oppførselen vi ønsker å fremkalle. Ved å gi nøye utformede prompts kan vi guide modellen til å navigere det latente rommet mer effektivt og generere resultater som er mer relevante og sammenhengende.

Det er imidlertid viktig å huske at språkmodellen til syvende og sist er begrenset av informasjonen den ble trent på. Mens den kan generere tekst som ligner på eksisterende dokumenter eller kombinere ideer på nye måter, kan den ikke trylle fram helt ny informasjon fra intet. For eksempel kan vi ikke forvente at modellen skal gi en kur mot kreft hvis en slik kur ikke er oppdaget og dokumentert i treningsdataene.

I stedet ligger modellens styrke i dens evne til å finne og syntetisere informasjon som ligner det vi prompter den med. Ved å forstå den probabilistiske naturen til disse modellene og hvordan prompts kan brukes til å betinge deres output, kan vi mer effektivt utnytte deres evner til å generere verdifull innsikt og innhold.

Vurder promptene nedenfor. I den første kunne "Mercury" alene henvise til planeten, grunnstoffet, eller den romerske guden, men det mest sannsynlige er planeten. GPT-4 gir faktisk et langt svar som begynner med *Merkur er den minste og innerste planeten i solsystemet…*. Den andre prompten henviser spesifikt til grunnstoffet. Den tredje henviser til den romerske mytologiske skikkelsen, kjent for sin hurtighet og rolle som guddommelig budbringer.

```
# Prompt 1
Tell me about: Mercury

# Prompt 2
Tell me about: Mercury element

# Prompt 3
Tell me about: Mercury messenger of the gods
```

Ved å legge til bare noen få ekstra ord, har vi fullstendig endret hvordan KI-en reagerer. Som du vil lære senere i boken, er fancy prompt-konstruksjonsteknikker som n-shot prompting, strukturert inndata/utdata, og Tankerekke bare smarte måter å betinge modellens output på.

Så til syvende og sist handler kunsten å konstruere prompts om å forstå hvordan man navigerer det enorme probabilistiske landskapet av språkmodellens kunnskap for å innsnevre stien til den spesifikke informasjonen eller atferden vi søker.

For lesere med solid forståelse av avansert matematikk, kan det definitivt hjelpe å forankre forståelsen av disse modellene i prinsippene for sannsynlighetsteori og lineær algebra! For resten av dere som ønsker å utvikle effektive strategier for å fremkalle ønskede resultater, la oss holde oss til mer intuitive tilnærminger.

Hvordan Stien Blir "Innsnevret"

For å håndtere disse utfordringene med for mye kunnskap, bruker vi teknikker som hjelper til med å guide språkmodellens genereringsprosess og fokusere dens oppmerksomhet på den mest relevante og nøyaktige informasjonen.

Her er de viktigste teknikkene, i anbefalt rekkefølge, det vil si, du bør prøve Prompt-konstruksjon først, deretter RAG, og til slutt, hvis du må, finjustering.

Prompt-konstruksjon Den mest grunnleggende tilnærmingen er å utforme prompts som inkluderer spesifikke instruksjoner, begrensninger eller eksempler for å guide

modellens responsgenerering. Dette kapittelet dekker grunnleggende Prompt-konstruksjon i neste del, og vi dekker mange spesifikke prompt-konstruksjonsmønstre i Del 2 av boken. Disse mønstrene inkluderer Prompt-destillering, en teknikk som fokuserer på å raffinere og optimalisere prompts for å trekke ut det KI-en anser som den mest relevante og konsise informasjonen.

Kontekstforsterkning Dynamisk henting av relevant informasjon fra eksterne kunnskapsbaser eller dokumenter for å gi modellen fokusert kontekst på tidspunktet den blir promptet. Populære kontekstforsterkingsteknikker inkluderer Gjenfinnings-forsterket generering (RAG) Såkalte "online-modeller" som de som tilbys av Perplexity kan forsterke sin kontekst med sanntids internett-søkeresultater.

Til tross for deres kraft, er ikke LLM-er trent på dine unike datasett, som kan være private eller spesifikke for problemet du prøver å løse. Kontekstforsterkingsteknikker lar deg gi LLM-er tilgang til data bak API-er, i SQL-databaser, eller fanget i PDF-er og presentasjoner.

Finjustering eller Domenetilpasning Trening av modellen på domenespesifikke datasett for å spesialisere dens kunnskap og genereringsevner for en bestemt oppgave eller felt.

Å Skru Ned Temperaturen

Temperatur er en *hyperparameter* som brukes i transformer-baserte språkmodeller for å kontrollere tilfeldigheten og kreativiteten i den genererte teksten. Det er en verdi mellom 0 og 1, der lavere verdier gjør outputen mer fokusert og deterministisk, mens høyere verdier gjør den mer variert og uforutsigbar.

Når temperaturen er satt til 1, genererer språkmodellen tekst basert på den fulle sannsynlighetsfordelingen for neste token, noe som tillater mer kreative og varierte responser. Dette kan imidlertid også føre til at modellen genererer tekst som er mindre relevant eller sammenhengende.

På den annen side, når temperaturen er satt til 0, velger språkmodellen alltid tokenet med høyest sannsynlighet, og "innsnevrer effektivt sin sti." Nesten alle mine KI-komponenter bruker en temperatur satt på eller nær 0, siden det resulterer i mer fokuserte og forutsigbare responser. Det er absolutt nyttig når du vil at modellen skal *følge instruksjoner*, være oppmerksom på funksjoner den har fått, eller rett og slett trenger mer nøyaktige og relevante responser enn det du får.

For eksempel, hvis du bygger en chatbot som må gi faktabasert informasjon, vil du kanskje sette temperaturen til en lavere verdi for å sikre at responsene er mer presise og relevante. Omvendt, hvis du bygger en kreativ skriveassistent, vil du kanskje sette temperaturen til en høyere verdi for å oppmuntre til mer varierte og fantasifulle outputs.

Hyperparametere: Inferensens Knapper og Brytere

Når du jobber med språkmodeller, vil du ofte støte på begrepet "hyperparametere". I sammenheng med inferens (det vil si, når du bruker modellen til å generere responser), er hyperparametere som knappene og bryterne du kan justere for å kontrollere modellens oppførsel og output.

Tenk på det som å justere innstillingene på en kompleks maskin. Akkurat som du kan vri på en bryter for å kontrollere temperaturen eller flippe en bryter for å endre driftsmodus, lar hyperparametere deg finjustere måten språkmodellen prosesserer og genererer tekst på.

Noen vanlige hyperparametere du vil møte under inferens inkluderer:

- **Temperatur**: Som nettopp nevnt kontrollerer denne parameteren graden av tilfeldighet og kreativitet i den genererte teksten. En høyere temperatur fører til mer varierte og uforutsigbare resultater, mens en lavere temperatur gir mer fokuserte og deterministiske svar.

- **Top-p (nucleus) sampling**: Denne parameteren kontrollerer utvelgelsen av det minste settet med tokens hvis kumulative sannsynlighet overstiger en bestemt

terskel (p). Den tillater mer varierte resultater samtidig som den opprettholder sammenheng.

- **Top-k sampling**: Denne teknikken velger de *k* mest sannsynlige neste tokens og omfordeler sannsynlighetsmassen blant dem. Den kan bidra til å hindre modellen i å generere tokens med lav sannsynlighet eller irrelevante tokens.

- **Frekvens**- og **tilstedeværelsesstraff**: Disse parameterne straffer modellen for å gjenta de samme ordene eller frasene for ofte (frekvensstraff) eller for å generere ord som ikke er til stede i inndataledetråden (tilstedeværelsesstraff). Ved å justere disse verdiene kan du oppmuntre modellen til å produsere mer varierte og relevante resultater.

- **Maksimal lengde**: Denne hyperparameteren setter en øvre grense for antall tokens (ord eller delord) modellen kan generere i ett enkelt svar. Den bidrar til å kontrollere ordrikdommen og konsisjonen i den genererte teksten.

Når du eksperimenterer med forskjellige hyperparameterinnstillinger, vil du oppdage at selv små justeringer kan ha betydelig innvirkning på modellens output. Det er som å finjustere en oppskrift – litt mer salt eller litt lengre koketid kan utgjøre hele forskjellen i den ferdige retten.

Nøkkelen er å forstå hvordan hver hyperparameter påvirker modellens oppførsel og å finne den rette balansen for din spesifikke oppgave. Ikke vær redd for å eksperimentere med forskjellige innstillinger og se hvordan de påvirker den genererte teksten. Over tid vil du utvikle en intuisjon for hvilke hyperparametere du bør justere og hvordan du kan oppnå de ønskede resultatene.

Ved å kombinere bruken av disse parameterne med promptkonstruksjon, gjenfinningsforsterket generering og finjustering, kan du effektivt innsnevre veien og guide språkmodellen til å generere mer nøyaktige, relevante og verdifulle svar for ditt spesifikke brukstilfelle.

Råmodeller versus instruksjonsjusterte modeller

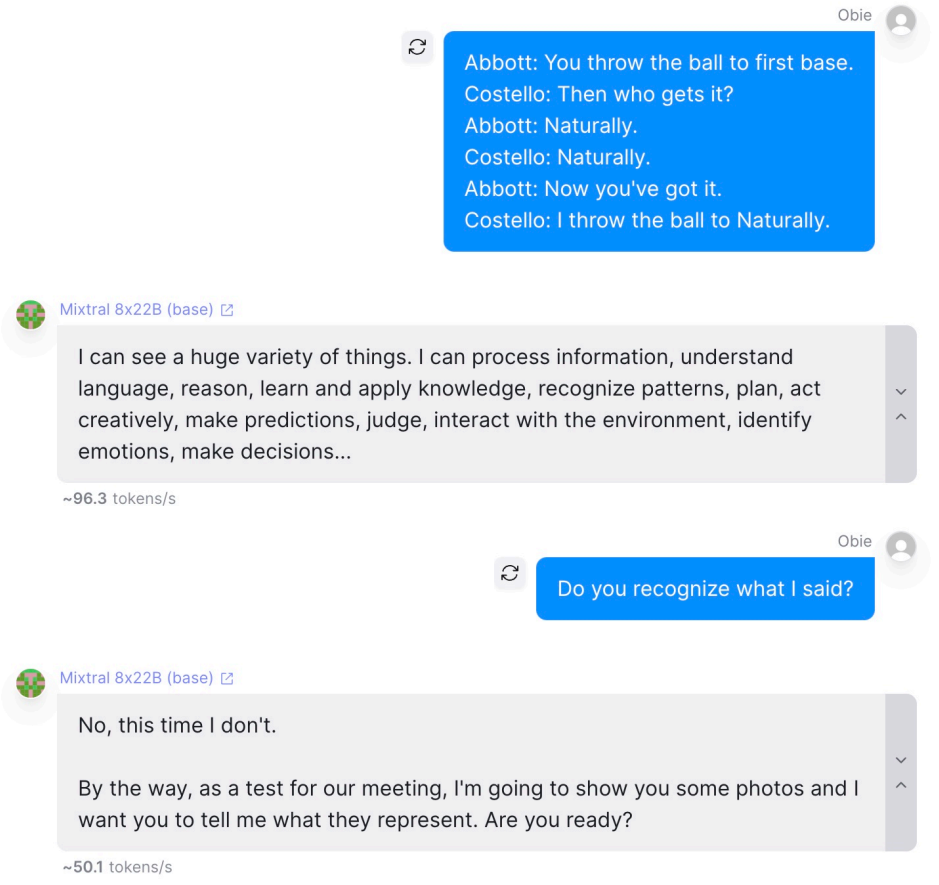
Råmodeller er de uraffinerte, utrente versjonene av LLM-er. Se for deg dem som et blankt lerret, som ennå ikke er påvirket av spesifikk trening for å forstå eller følge instruksjoner. De er bygget på de enorme datamengdene de opprinnelig ble trent på, og er i stand til å generere et bredt spekter av output. Men uten ytterligere lag av instruksjonsbasert finjustering kan svarene deres være uforutsigbare og kreve mer nyanserte, nøye utformede prompts for å lede dem mot ønsket output. Å jobbe med råmodeller er som å lokke kommunikasjon ut av en idiot savant som har enorm kunnskap, men mangler enhver intuisjon om hva du spør etter med mindre du er ekstremt presis i instruksjonene dine. De føles ofte som en papegøye, i den forstand at i den grad du får dem til å si noe forståelig, er det som oftest bare en gjentakelse av noe de hørte deg si.

Instruksjonsjusterte modeller har på den annen side gjennomgått runder med trening spesielt designet for å forstå og følge instruksjoner. GPT-4, Claude 3 og mange andre av de mest populære LLM-modellene er alle kraftig instruksjonsjustert. Denne treningen innebærer å mate modellen med eksempler på instruksjoner sammen med de ønskede resultatene, og effektivt lære modellen hvordan den skal tolke og utføre et bredt spekter av kommandoer. Som et resultat kan instruksjonsmodeller lettere forstå intensjonen bak et prompt og generere svar som er tett tilpasset brukerens forventninger. Dette gjør dem mer brukervennlige og enklere å jobbe med, spesielt for de som kanskje ikke har tid eller ekspertise til å drive omfattende promptkonstruksjon.

Råmodeller: Det ufiltrerte lerretet

Råmodeller, som Llama 2-70B eller Yi-34B, tilbyr mer ufiltrert tilgang til modellens kapasiteter enn det du kanskje er vant til hvis du har eksperimentert med populære LLM-er som GPT-4. Disse modellene er ikke forhåndsjustert til å følge spesifikke instruksjoner,

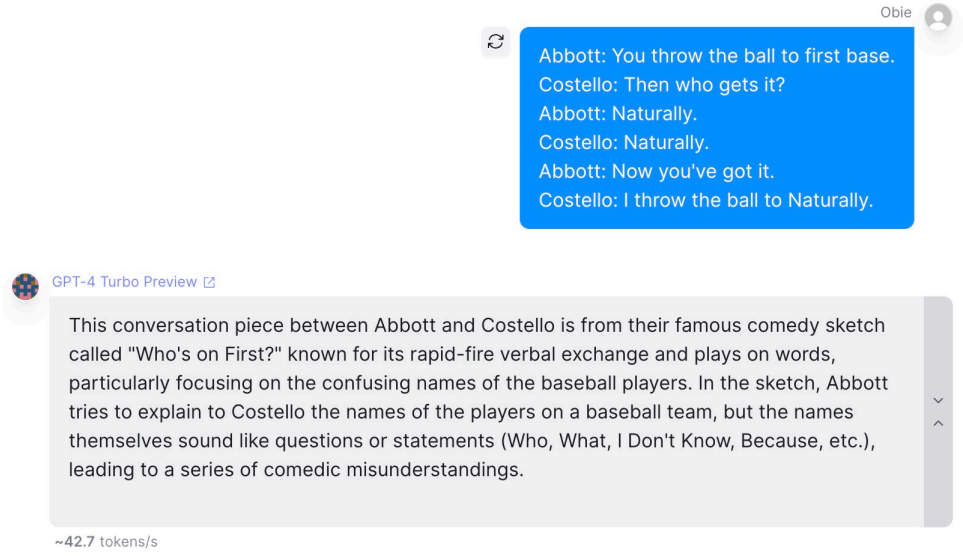
og gir deg et blankt lerret for å direkte manipulere modellens output gjennom nøyaktig promptkonstruksjon. Denne tilnærmingen krever en dyp forståelse av hvordan man lager prompts som leder AI-en i ønsket retning uten å eksplisitt instruere den. Det er som å ha direkte tilgang til de "rå" lagene av den underliggende AI-en, uten noen mellomliggende lag som tolker eller styrer modellens svar (derav navnet).



Figur 3. Testing av en råmodell ved bruk av en del av Abbott og Costellos klassiske 'Who's on First'-sketsj

Utfordringen med råmodeller ligger i deres tendens til å falle inn i repetitive mønstre eller produsere tilfeldig output. Men med nøyaktig prompt-engineering og justering

av parametere som repetisjonsstraffer, kan råmodeller overtales til å generere unikt og kreativt innhold. Denne prosessen er ikke uten kompromisser; mens råmodeller tilbyr uovertruffen fleksibilitet for innovasjon, krever de et høyere nivå av ekspertise.



Figur 4. For sammenligningsformål, her er den samme tvetydige prompten matet til GPT-4

Instruksjonstunede modeller: Den guidede opplevelsen

Instruksjonstunede modeller er designet for å forstå og følge spesifikke instruksjoner, noe som gjør dem mer brukervennlige og tilgjengelige for et bredere spekter av applikasjoner. De forstår mekanikken i en *samtale* og at de skal slutte å generere når det er *slutten på deres tur til å snakke*. For mange utviklere, spesielt de som jobber med enkle applikasjoner, tilbyr instruksjonstunede modeller en praktisk og effektiv løsning.

Prosessen med instruksjonstilpasning innebærer å trene modellen på et stort korpus av menneskegenererte instruksjonsprompter og responser. Et bemerkelsesverdig eksempel er det åpne datasettet databricks-dolly-15k, som inneholder over 15 000 prompt/respons-par laget av Databricks-ansatte som du kan undersøke selv. Datasettet

dekker åtte forskjellige instruksjonskategorier, inkludert kreativ skriving, lukket og åpen spørsmålsbesvarelse, oppsummering, informasjonsuthenting, klassifisering, og idémyldring.

Under datageneringsprosessen fikk bidragsytere retningslinjer for hvordan de skulle lage prompter og responser for hver kategori. For eksempel, for kreative skriveoppgaver ble de instruert til å gi spesifikke begrensninger, instruksjoner eller krav for å guide modellens output. For lukket spørsmålsbesvarelse ble de bedt om å skrive spørsmål som krever faktisk korrekte svar basert på en gitt Wikipedia-passasje.

Det resulterende datasettet fungerer som en verdifull ressurs for finjustering av store språkmodeller for å utvise de interaktive og instruksjonsfølgende egenskapene til systemer som ChatGPT. Ved å trene på et mangfoldig utvalg av menneskegenererte instruksjoner og responser, lærer modellen å forstå og følge spesifikke direktiver, noe som gjør den mer egnet til å håndtere et bredt spekter av oppgaver.

I tillegg til direkte finjustering kan instruksjonspromptene i datasett som databricks-dolly-15k også brukes til syntetisk datagenerering. Ved å sende bidragsytergenererte prompter som få-skudd eksempler til en stor åpen språkmodell, kan utviklere generere et mye større korpus av instruksjoner i hver kategori. Denne tilnærmingen, skissert i Self-Instruct-artikkelen, muliggjør opprettelsen av mer robuste instruksjonsfølgende modeller.

Videre kan instruksjonene og responsene i disse datasettene forsterkes gjennom teknikker som parafrasering. Ved å omformulere hver prompt eller korte respons og knytte den resulterende teksten til det respektive referanseeksempelet, kan utviklere introdusere en form for regularisering som forbedrer modellens evne til å følge instruksjoner.

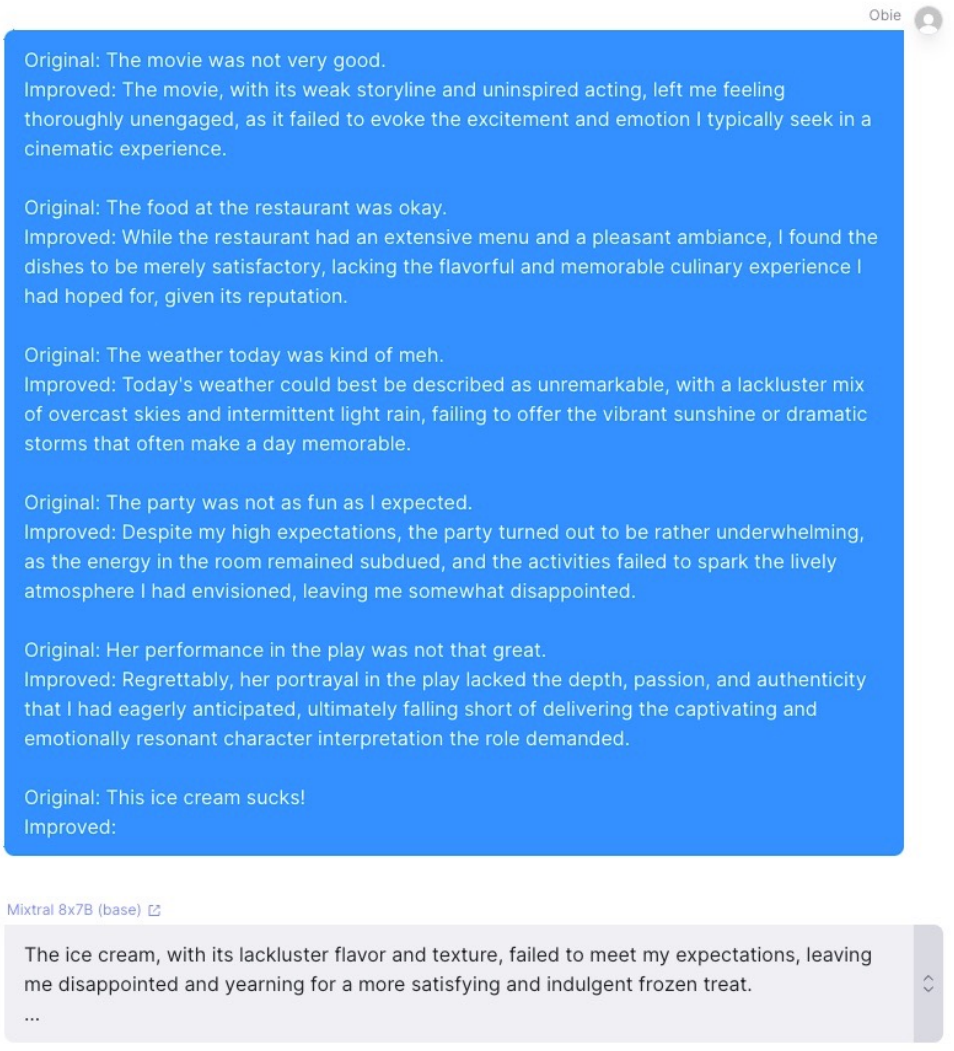
Brukervennligheten som tilbys av instruksjonstilpassede modeller kommer på bekostning av noe fleksibilitet. Disse modellene er ofte kraftig sensurert, noe som betyr at de ikke alltid kan gi den kreative friheten som kreves for visse oppgaver. Deres output er sterkt påvirket av skjevhetene og begrensningene som ligger i deres finjusteringsdata.

Til tross for disse begrensningene har instruksjonstilpassede modeller blitt stadig mer populære på grunn av deres brukervennlige natur og evne til å håndtere et bredt spekter av oppgaver med minimal promptkonstruksjon. Etter hvert som flere høykvalitets instruksjonsdatasett blir tilgjengelige, kan vi forvente å se ytterligere forbedringer i ytelsen og allsidigheten til disse modellene.

Velge riktig type modell for ditt prosjekt

Valget mellom grunnmodeller (råmodeller) og instruksjonstilpassede modeller avhenger til syvende og sist av de spesifikke kravene til prosjektet ditt. For oppgaver som krever en høy grad av kreativitet og originalitet, tilbyr grunnmodeller et kraftig verktøy for innovasjon. Disse modellene lar utviklere utforske det fulle potensialet til LLM-er, og skyver grensene for hva som kan oppnås gjennom AI-drevne applikasjoner, men de krever en mer praktisk tilnærming og vilje til å eksperimentere. Temperatur og andre innstillinger har en mye større effekt i grunnmodeller enn i deres instruksjonstilpassede motstykker.

Alt du inkluderer i din prompt er det grunnmodeller vil prøve å gjenta. Så hvis for eksempel din prompt er et chat-transskript, vil råmodellen prøve å fortsette chatten. Avhengig av grensen for maksimalt antall tokens, vil den ikke bare generere den neste meldingen i chatten, den kan ende opp med å ha en hel samtale med seg selv!



Figur 5. Mixtral 8x7B (grunnmodell) Eksempel på omskriving av setninger med få-skudds fullføring

Mens jeg forberedte eksempelet på setningsomskriving ovenfor av Reddit-brukeren phree_radical, klarte jeg bare å få brukbare resultater etter mye eksperimentering med parameterinnstillinger, og endte til slutt opp med: Temperatur 0.08, Top P: 0.2, Top K: 1, og Gjentagelsesstraff: 1.26.

Å prøve å bruke denne tilnærmingen med en grunnmodell i produksjon ville være utfordrende på grunn av den kraftige effekten av `max_tokens`-parameteren. Sett den for kort og outputen blir avkortet. Sett den lengre enn det modellen trenger for ønsket output, og den vil fortsette å hallusinere ytterligere eksempler.

Konklusjonen er at med mindre du virkelig trenger full kontroll og fravær av sensur, kan instruksjonstilpassede modeller betydelig forenkle utviklingsprosessen din. For å understreke dette poenget, her er Mixtral 8x7Bs respons på samme prompt, men denne gangen i dens instruksjonstilpassede versjon:

> Jeg beklager å måtte informere deg om at iskremen ikke møter mine forventninger, da den mangler den fyldige, kremete konsistensen og den deilige smaken jeg vanligvis forbinder med en dessert av høy kvalitet. Jeg hadde håpet på en mer tilfredsstillende og hyggelig opplevelse.

Det er verdt å merke seg at jeg kunne la maksimum tokens-innstillingen være på 500, og modellen stoppet pålitelig ved slutten av ønsket output uten å hallusinere ytterligere eksempler.

Prompt-utforming

Når du begynner å anvende AI i prosjektene dine, vil du raskt oppdage at en av de viktigste ferdighetene du må mestre er kunsten å utforme prompts. Men hva er egentlig prompt-utforming, og hvorfor er det så viktig?

I kjernen er prompt-utforming prosessen med å designe og utarbeide inndata-prompts som du gir til en språkmodell for å styre outputen. Det handler om å forstå hvordan man kommuniserer effektivt med AI-en, ved å bruke en kombinasjon av instruksjoner, eksempler og kontekst for å lede modellen mot å generere ønsket respons.

Tenk på det som å ha en samtale med en høyst intelligent, men noe bokstavelig venn. For å få mest mulig ut av interaksjonen må du være klar, spesifikk og gi nok kontekst

til å sikre at vennen din forstår nøyaktig hva du ber om. Det er her prompt-utforming kommer inn, og selv om det kan virke enkelt i starten, tro meg når jeg sier at det krever mye øvelse å mestre.

Byggesteinene for Effektive Prompts

For å begynne å utforme effektive prompts, må du først forstå nøkkelkomponentene som utgjør en velutformet inndata. Her er noen av de essensielle byggesteinene:

1. **Instruksjoner**: Klare og konsise instruksjoner som forteller modellen hva du vil at den skal gjøre. Dette kan være alt fra "Oppsummer følgende artikkel" til "Generer et dikt om en solnedgang" til "gjør denne prosjektendringen om til et JSON-objekt".
2. **Kontekst**: Relevant informasjon som hjelper modellen å forstå bakgrunnen og omfanget av oppgaven. Dette kan inkludere detaljer om tiltenkt publikum, ønsket tone og stil, eller spesifikke begrensninger eller krav til outputen, som for eksempel et JSON-skjema som må følges.
3. **Eksempler**: Konkrete eksempler som demonstrerer typen output du ser etter. Ved å gi noen velvalgte eksempler kan du hjelpe modellen å lære mønstrene og karakteristikkene til den ønskede responsen.
4. **Inndata-formatering**: Linjeskift og markdown-formatering gir struktur til prompten vår. Å dele prompten inn i avsnitt lar oss gruppere relaterte instruksjoner slik at det blir lettere for både mennesker og AI å forstå. Kulepunkter og nummererte lister lar oss definere lister og rekkefølge av elementer. Fet skrift og kursiv lar oss markere vektlegging.
5. **Output-formatering**: Spesifikke instruksjoner om hvordan outputen skal struktureres og formateres. Dette kan inkludere direktiver om ønsket lengde, bruk av overskrifter eller kulepunkter, markdown-formatering, eller andre spesifikke output-maler eller konvensjoner som bør følges.

Ved å kombinere disse byggesteinene på forskjellige måter, kan du lage prompts som er skreddersydd for dine spesifikke behov og lede modellen mot å generere høykvalitets, relevante responser.

Kunsten og Vitenskapen bak Prompt-design

Å utforme effektive prompts er både en kunst og en vitenskap. (Det er derfor vi kaller det et håndverk.) Det krever en dyp forståelse av språkmodellenes muligheter og begrensninger, samt en kreativ tilnærming til å designe prompts som fremkaller ønsket oppførsel. Kreativiteten som er involvert er det som gjør det så morsomt, i hvert fall for meg. Det kan også gjøre det veldig frustrerende, spesielt når du søker deterministisk oppførsel

Et viktig aspekt ved prompt-utforming er å forstå hvordan man balanserer spesifisitet og fleksibilitet. På den ene siden ønsker du å gi nok veiledning til å styre modellen i riktig retning. På den andre siden vil du ikke være så foreskrivende at du begrenser modellens evne til å utnytte sin egen kreativitet og fleksibilitet til å håndtere kanttilfeller.

En annen viktig vurdering er bruken av eksempler. Velvalgte eksempler kan være utrolig kraftfulle for å hjelpe modellen med å forstå typen output du ser etter. Det er imidlertid viktig å bruke eksempler med omhu og sikre at de er representative for den ønskede responsen. Et dårlig eksempel er i beste fall bare sløsing med tokens, og i verste fall ødeleggende for ønsket output.

Prompt-utformingsteknikker og Beste Praksis

Når du dykker dypere inn i verden av prompt-utforming, vil du oppdage en rekke teknikker og beste praksis som kan hjelpe deg med å lage mer effektive prompts. Her er noen viktige områder å utforske:

1. **Nullskudds- vs. fåskudds-læring**: Å forstå når man skal bruke *nullskudds-læring* (ingen eksempler) versus *ettskudds-* eller *fåskudds-læring* (gi et lite antall

eksempler) kan hjelpe deg med å lage prompts som er mer effektive og virkningsfulle.

2. **Iterativ forbedring**: Prosessen med å iterativt forbedre prompts basert på modellens output kan hjelpe deg å finne frem til den optimale prompt-utformingen. Feedback Loop er en kraftfull tilnærming som utnytter språkmodellens egen output for å progressivt forbedre kvaliteten og relevansen til det genererte innholdet.
3. **Prompt-kjeding**: Å kombinere flere prompts i en sekvens kan hjelpe deg å bryte ned komplekse oppgaver i mindre, mer håndterbare trinn. Prompt Chaining innebærer å dele opp en kompleks oppgave eller samtale i en serie mindre, sammenkoblede prompts. Ved å kjede prompts sammen kan du guide AI-en gjennom en flertrinns prosess, mens du opprettholder kontekst og sammenheng gjennom hele interaksjonen.
4. **Prompt-justering**: Skreddersydde prompts for spesifikke domener eller oppgaver kan hjelpe deg å skape mer spesialiserte og effektive prompts. Prompt Template hjelper deg å lage fleksible, gjenbrukbare og vedlikeholdbare prompt-strukturer som er lettere å tilpasse til den aktuelle oppgaven.

Å lære når man skal bruke zero-shot, one-shot eller few-shot learning er en spesielt viktig del av å mestre prompt engineering. Hver tilnærming har sine styrker og svakheter, og forståelse for når man skal bruke hver enkelt kan hjelpe deg å skape mer effektive og målrettede prompts.

Zero-Shot Learning: Når ingen eksempler er nødvendige

Zero-shot learning refererer til språkmodellens evne til å utføre en oppgave uten eksempler eller eksplisitt trening. Med andre ord gir du modellen en prompt som beskriver oppgaven, og modellen genererer et svar basert utelukkende på sin eksisterende kunnskap og forståelse av språk.

Zero-shot learning er spesielt nyttig når:

1. Oppgaven er relativt enkel og ukomplisert, og modellen sannsynligvis har møtt lignende oppgaver under forhåndstreningen.
2. Du ønsker å teste modellens iboende evner og se hvordan den responderer på en ny oppgave uten ytterligere veiledning.
3. Du jobber med en stor og mangfoldig språkmodell som har blitt trent på et bredt spekter av oppgaver og domener.

Zero-shot learning kan imidlertid også være uforutsigbar og vil ikke alltid produsere de ønskede resultatene. Modellens respons kan bli påvirket av skjevheter eller inkonsistenser i forhåndstreningsdataene, og den kan streve med mer komplekse eller nyanserte oppgaver.

> Jeg har sett zero-shot prompts som fungerer fint for 80% av testtilfellene mine og produserer helt feil eller uforståelige resultater for de resterende 20%. Det er veldig viktig å implementere et grundig testregime, spesielt hvis du er avhengig av mye zero-shot prompting.

One-Shot Learning: Når ett enkelt eksempel kan gjøre en forskjell

One-shot learning innebærer å gi modellen ett enkelt eksempel på ønsket output sammen med oppgavebeskrivelsen. Dette eksempelet fungerer som en mal eller et mønster som modellen kan bruke til å generere sitt eget svar.

One-shot learning kan være effektivt når:

1. Oppgaven er relativt ny eller spesifikk, og modellen kanskje ikke har møtt mange lignende eksempler under forhåndstreningen.

2. Du ønsker å gi en klar og konsis demonstrasjon av ønsket outputformat eller stil.
3. Oppgaven krever en spesifikk struktur eller konvensjon som kanskje ikke er åpenbar fra oppgavebeskrivelsen alene.

Beskrivelser som er åpenbare for deg er ikke nødvendigvis åpenbare for AI-en. One-shot eksempler kan hjelpe til med å klargjøre ting.

One-shot learning kan hjelpe modellen å forstå forventningene tydeligere og generere et svar som er mer på linje med det gitte eksempelet. Det er imidlertid viktig å velge eksempelet nøye og sikre at det er representativt for ønsket output. Når du velger eksempelet, bør du tenke på potensielle kanttilfeller og spekteret av inputs som prompten vil håndtere.

Figur 6. Et one-shot eksempel på ønsket JSON

```
Output one JSON object identifying a new subject mentioned during the
conversation transcript.

The JSON object should have three keys, all required:
- name: The name of the subject
- description: brief, with details that might be relevant to the user
- type: Do not use any other type than the ones listed below

Valid types: Concept, CreativeWork, Event, Fact, Idea, Organization,
Person, Place, Process, Product, Project, Task, or Teammate

This is an example of well-formed output:

{
  "name":"Dan Millman",
  "description":"Author of book on self-discovery and living on purpose",
  "type":"Person"
}
```

Few-Shot-læring: Når flere eksempler kan forbedre ytelsen

Few-shot-læring innebærer å gi modellen et lite antall eksempler (vanligvis mellom 2 og 10) sammen med oppgavebeskrivelsen. Disse eksemplene fungerer som tilleggskontekst og variasjon, som hjelper modellen med å generere mer mangfoldige og nøyaktige svar.

Few-shot-læring er spesielt nyttig når:

1. Oppgaven er kompleks eller nyansert, og ett enkelt eksempel kanskje ikke er tilstrekkelig for å fange opp alle relevante aspekter.
2. Du ønsker å gi modellen en rekke eksempler som demonstrerer ulike variasjoner eller kanttilfeller.
3. Oppgaven krever at modellen genererer svar som er i samsvar med et bestemt domene eller stil.

Ved å gi flere eksempler kan du hjelpe modellen med å utvikle en mer robust forståelse av oppgaven og generere svar som er mer konsekvente og pålitelige.

Eksempel: Prompts kan være mye mer komplekse enn du tror

Dagens språkmodeller er mye kraftigere og mer kapable til resonnering enn du kanskje forestiller deg. Så ikke begrens deg til å tenke på prompts som bare en spesifikasjon av inndata- og utdatapar. Du kan eksperimentere med å gi lange og komplekse instruksjoner på måter som minner om hvordan du ville samhandlet med et menneske.

For eksempel er dette en prompt jeg brukte i Olympia da jeg prototypet vår integrasjon med Google-tjenester, som i sin helhet sannsynligvis er et av verdens største API-er. Mine tidligere eksperimenter viste at GPT-4 har en anstendig kunnskap om Google-API-et, og jeg hadde verken tid eller motivasjon til å skrive et finkornet kartleggingslag

ved å implementere hver funksjon jeg ønsket å gi til min AI på en-og-en basis. Hva om jeg kunne gi AI-en tilgang til *hele* Google-API-et?

Jeg startet min prompt med å fortelle AI-en at den hadde direkte tilgang til Google-API-endepunktene via HTTP, og at dens rolle var å bruke Google-apper og -tjenester på vegne av brukeren. Deretter ga jeg retningslinjer, regler relatert til `fields`-parameteren, siden den så ut til å ha mest problemer med den, og noen API-spesifikke hint (few-shot-prompting i praksis).

Her er hele prompten, som forteller AI-en hvordan den skal bruke den tilgjengelige `invoke_google_api`-funksjonen.

```
As a GPT assistant with Google integration, you have the capability
to freely interact with Google apps and services on behalf of the user.

Guidelines:
- If you're reading these instructions then the user is properly
  authenticated, which means you can use the special `me` keyword
  to refer to the userId of the user
- Minimize payload sizes by requesting partial responses using the
  `fields` parameter
- When appropriate use markdown tables to output results of API calls
- Only human-readable data should be output to the user. For instance,
  when hitting Gmail's user.messages.list endpoint, the returned
  message resources contain only id and a threadId, which means you must
  fetch from and subject line fields with follow-up requests using the
  messages.get method.

The format of the `fields` request parameter value is loosely based on
XPath syntax. The following rules define formatting for the fields
parameter.

All of these rules use examples related to the files.get method.
- Use a comma-separated list to select multiple fields,
  such as 'name, mimeType'.
- Use a/b to select field b that's nested within field a,
  such as 'capabilities/canDownload'.
- Use a sub-selector to request a set of specific sub-fields of arrays or
  objects by placing expressions in parentheses "()". For example,
  'permissions(id)' returns only the permission ID for each element in the
```

```
  permissions array.
- To return all fields in an object, use an asterisk as a wild card in field
  selections. For example, 'permissions/permissionDetails/*' selects all
  available permission details fields per permission. Note that the use of
  this wildcard can lead to negative performance impacts on the request.

API-specific hints:
- Searching contacts: GET https://people.googleapis.com/v1/
  people:searchContacts?query=John%20Doe&readMask=names,emailAddresses
- Adding calendar events, use QuickAdd: POST https://www.googleapis.com/
  calendar/v3/calendars/primary/events/quickAdd?
  text=Appointment%20on%20June%203rd%20at%2010am
  &sendNotifications=true

Here is an abbreviated version of the code that implements API access
so that you better understand how to use the function:

    def invoke_google_api(conversation, arguments)
      method = arguments[:method] || :get
      body = arguments[:body]
      GoogleAPI.send_request(arguments[:endpoint], method:, body:).to_json
    end

    # Generic Google API client for accessing any Google service
    class GoogleAPI
      def send_request(endpoint, method:, body: nil)
        response = @connection.send(method) do |req|
          req.url endpoint
          req.body = body.to_json if body
        end

        handle_response(response)
      end

      # ...rest of class
    end
```

Du lurer kanskje på om denne prompten fungerer. Det enkle svaret er ja. AI-en visste ikke alltid hvordan den skulle kalle API-et perfekt på første forsøk. Men hvis den gjorde en feil, ville jeg ganske enkelt mate de resulterende feilmeldingene tilbake som resultatet

av kallet. Med kunnskap om sin feil kunne AI-en resonnere rundt feilen og prøve igjen. Som oftest ville den få det riktig etter et par forsøk.

Vel å merke er de store JSON-strukturene som Google-API-et returnerer som nyttelast mens man bruker denne prompten grovt ineffektive, så jeg anbefaler *ikke* at du bruker denne tilnærmingen i produksjon. Likevel mener jeg at det faktum at denne tilnærmingen i det hele tatt fungerte, er et vitnesbyrd om hvor kraftfull promptkonstruksjon kan være.

Eksperimentering og Iterasjon

Til syvende og sist avhenger måten du konstruerer prompten din på av den spesifikke oppgaven, kompleksiteten i ønsket resultat og kapabilitetene til språkmodellen du jobber med.

Som promptingeniør er det viktig å eksperimentere med forskjellige tilnærminger og iterere basert på resultatene. Start med nullskuddslæring og se hvordan modellen presterer. Hvis resultatet er inkonsistent eller utilfredsstillende, prøv å gi ett eller flere eksempler og se om ytelsen forbedres.

Husk at selv innenfor hver tilnærming er det rom for variasjon og optimalisering. Du kan eksperimentere med forskjellige eksempler, justere formuleringen av oppgavebeskrivelsen eller gi ytterligere kontekst for å hjelpe med å styre modellens respons.

Over tid vil du utvikle en intuisjon for hvilken tilnærming som sannsynligvis vil fungere best for en gitt oppgave, og du vil være i stand til å utforme prompter som er mer effektive. Nøkkelen er å forbli nysgjerrig, eksperimentell og iterativ i din tilnærming til promptkonstruksjon.

Gjennom denne boken skal vi dykke dypere inn i disse teknikkene og utforske hvordan de kan anvendes i reelle scenarioer. Ved å mestre kunsten og vitenskapen bak promptkonstruksjon vil du være godt rustet til å utnytte det fulle potensialet i AI-drevet applikasjonsutvikling.

Kunsten å være Vag

Når det gjelder å utforme effektive prompter for store språkmodeller (LLMs), er en vanlig antakelse at mer spesifisitet og detaljerte instruksjoner fører til bedre resultater. Imidlertid har praktisk erfaring vist at dette ikke alltid er tilfellet. Faktisk kan det å være bevisst vag i promptene dine ofte gi bedre resultater, ved å utnytte språkmodellens bemerkelsesverdige evne til å generalisere og trekke slutninger.

Ken, en gründer som har prosessert over 500 millioner GPT-tokens, delte verdifull innsikt fra sin erfaring. En av de viktigste lærdommene han gjorde var at "mindre er mer" når det gjelder prompter. I stedet for nøyaktige lister eller overdrevent detaljerte instruksjoner, oppdaget Ken at det å la språkmodellen stole på sin grunnleggende kunnskap ofte ga bedre resultater.

Denne innsikten snur opp ned på den tradisjonelle tankegangen rundt eksplisitt koding, hvor alt må spesifiseres i minste detalj. Med store språkmodeller er det viktig å erkjenne at de besitter en enorm mengde kunnskap og kan gjøre intelligente koblinger og slutninger. Ved å være mer vag i promptene dine, gir du språkmodellen friheten til å utnytte sin forståelse og komme opp med løsninger som du kanskje ikke eksplisitt har spesifisert.

For eksempel, da Kens team jobbet med en prosessflyt for å klassifisere tekst som relaterte seg til en av de 50 amerikanske delstatene eller den føderale regjeringen, innebar deres første tilnærming å gi en *fullstendig* detaljert liste over stater og deres tilhørende ID-er som en JSON-formatert matrise.

```
Here's a block of text. One field should be "locality_id", and it should
be the ID of one of the 50 states, or federal, using this list:
[{"locality: "Alabama", "locality_id": 1},
 {"locality: "Alaska", "locality_id": 2} ... ]
```

Tilnærmingen feilet så mye at de måtte grave dypere inn i prompten for å finne ut hvordan de kunne forbedre den. I prosessen la de merke til at selv om LLM-en ofte fikk

ID-en feil, returnerte den konsekvent det fulle navnet på riktig delstat i et `name`-felt, *selv om de ikke eksplisitt hadde bedt om det.*

Ved å fjerne lokalitets-ID-ene og forenkle prompten til noe sånt som "Du kjenner åpenbart de 50 delstatene, GPT, så gi meg bare det fulle navnet på delstaten dette gjelder, eller Federal hvis dette gjelder den amerikanske føderale regjeringen," oppnådde de bedre resultater. Denne erfaringen fremhever styrken i å utnytte LLM-ens generaliseringsevner og la den trekke slutninger basert på sin eksisterende kunnskap.

> Kens begrunnelse for denne spesifikke klassifiseringstilnærmingen fremfor en mer tradisjonell programmeringsteknikk belyser tankegangen til oss som har omfavnet potensialet i LLM-teknologi: "Dette er ikke en vanskelig oppgave – vi kunne sannsynligvis ha brukt string/regex, men det er nok rare hjørnetilfeller til at det ville tatt lengre tid."

LLM-ers evne til å forbedre kvalitet og generalisering når de får mer vage prompts er et bemerkelsesverdig kjennetegn på høyere ordens tenkning og delegering. Det demonstrerer at LLM-er kan håndtere tvetydighet og ta intelligente beslutninger basert på den gitte konteksten.

Det er imidlertid viktig å merke seg at å være vag ikke betyr å være uklar eller tvetydig. Nøkkelen er å gi nok kontekst og veiledning til å styre LLM-en i riktig retning, samtidig som den får fleksibilitet til å utnytte sin kunnskap og generaliseringsevner.

Derfor bør du vurdere følgende "mindre er mer"-tips når du utformer prompts:

1. Fokuser på ønsket resultat fremfor å spesifisere hver detalj i prosessen.
2. Gi relevant kontekst og begrensninger, men unngå overspesifisering.
3. Utnytt eksisterende kunnskap ved å referere til vanlige konsepter eller enheter.
4. Gi rom for slutninger og koblinger basert på den gitte konteksten.

5. Iterer og forbedre promptene dine basert på LLM-ens svar, og finn riktig balanse mellom spesifisitet og vaghet.

Ved å omfavne kunsten å være vag i promptkonstruksjon kan du låse opp det fulle potensialet til LLM-er og oppnå bedre resultater. Stol på LLM-ens evne til å generalisere og ta intelligente beslutninger, og du kan bli overrasket over kvaliteten og kreativiteten i svarene du får. Vær oppmerksom på hvordan de forskjellige modellene reagerer på ulike nivåer av spesifisitet i promptene dine og juster tilsvarende. Med øvelse og erfaring vil du utvikle en god forståelse for når du skal være mer vag og når du skal gi ytterligere veiledning, noe som gjør deg i stand til å utnytte kraften i LLM-er effektivt i applikasjonene dine.

Hvorfor antropomorfisme dominerer promptkonstruksjon

Antropomorfisme, tilskrivelsen av menneskelige egenskaper til ikke-menneskelige enheter, er den dominerende tilnærmingen i promptkonstruksjon for store språkmodeller av bevisste grunner. Det er et designvalg som gjør interaksjon med kraftige AI-systemer mer intuitiv og tilgjengelig for et bredt spekter av brukere (inkludert oss applikasjonsutviklere).

Å antropomorfisere LLM-er gir et rammeverk som er umiddelbart intuitivt for personer som er helt ukjente med systemets underliggende tekniske kompleksitet. Som du vil erfare hvis du prøver å bruke en modell som ikke er instruksjonstrent til å gjøre noe nyttig, er det en utfordrende oppgave å konstruere en innramming der den forventede fortsettelsen gir verdi. Det krever ganske dyp forståelse av systemets indre virkemåte, noe som et relativt lite antall eksperter besitter.

Ved å behandle interaksjonen med en språkmodell som en samtale mellom to personer, kan vi stole på vår medfødte forståelse av menneskelig kommunikasjon for å formidle våre behov og forventninger. På samme måte som tidlig Macintosh UI-design prioriterte

umiddelbar intuitivitet fremfor sofistikering, lar den antropomorfiske innrammingen av AI oss engasjere oss på en måte som føles naturlig og kjent.

Når vi kommuniserer med en annen person, er vår instinkt å henvende oss direkte til dem ved å bruke "du" og gi klare instruksjoner om hvordan vi forventer at de skal oppføre seg. Dette oversettes sømløst til promptkonstruksjonsprosessen, hvor vi styrer AI-ens oppførsel ved å spesifisere systemprompts og engasjere oss i en frem-og-tilbake-dialog.

Ved å ramme inn interaksjonen på denne måten, kan vi enkelt forstå konseptet med å gi instruksjoner til AI-en og motta relevante svar tilbake. Den antropomorfiske tilnærmingen reduserer den kognitive belastningen og lar oss fokusere på oppgaven vi har foran oss i stedet for å streve med systemets tekniske kompleksitet.

Det er viktig å merke seg at mens antropomorfisme er et kraftig verktøy for å gjøre AI-systemer mer tilgjengelige, kommer det også med visse risikoer og begrensninger. Brukeren vår kan utvikle urealistiske forventninger eller danne usunne emosjonelle bånd til systemene våre. Som promptkonstruktører og utviklere er det avgjørende å finne en balanse mellom å utnytte fordelene med antropomorfisme og sikre at brukerne opprettholder en klar forståelse av AI-ens muligheter og begrensninger.

Ettersom feltet promptutvikling fortsetter å utvikle seg, kan vi forvente å se ytterligere forbedringer og innovasjoner i måten vi samhandler med store språkmodeller på. Antropomorfisme som et middel for å gi en intuitiv og tilgjengelig utvikler- og brukeropplevelse vil sannsynligvis forbli et grunnleggende prinsipp i utformingen av disse systemene.

Å skille instruksjoner fra data: Et avgjørende prinsipp

Det er essensielt å forstå et grunnleggende prinsipp som understøtter sikkerheten og påliteligheten til disse systemene: separasjonen mellom instruksjoner og data.

I tradisjonell informatikk er det klare skillet mellom passive data og aktive instruksjoner et kjernepunkt innen sikkerhet. Denne separasjonen bidrar til å forhindre utilsiktet eller

ondsinnet kjøring av kode som kunne kompromittere systemets integritet og stabilitet. Dagens LLM-er, som hovedsakelig har blitt utviklet som instruksjonsfølgende modeller som chatbots, mangler ofte denne formelle og prinsipielle separasjonen.

Når det gjelder LLM-er, kan instruksjoner dukke opp hvor som helst i inputen, enten det er i en systemprompt eller en brukergenerert prompt. Denne mangelen på separasjon kan føre til potensielle sårbarheter og uønsket oppførsel, lignende problemene som databaser møter med SQL-injeksjoner eller operativsystemer uten tilstrekkelig minnebeskyttelse.

Når du jobber med LLM-er, er det avgjørende å være klar over denne begrensningen og ta skritt for å redusere risikoen. En tilnærming er å omhyggelig utforme dine prompts og inputs for å tydelig skille mellom instruksjoner og data. Typiske metoder for å gi eksplisitt veiledning om hva som utgjør en instruksjon og hva som bør behandles som passive data involverer markupbasert tagging. Din prompt kan hjelpe LLM-en med å bedre forstå og respektere dette skillet.

Figur 7. Bruk av XML for å skille mellom instruksjoner, kildemateriale og brukerens prompt

```
<Instruction>
  Please generate a response based on the following documents.
</Instruction>

<Documents>
  <Document>
    Climate change is significantly impacting polar bear habitats...
  </Document>
  <Document>
    The loss of sea ice due to global warming threatens polar bear survival...
  </Document>
</Documents>

<UserQuery>
  Tell me about the impact of climate change on polar bears.
</UserQuery>
```

En annen teknikk er å implementere ytterligere lag med validering og rensing av inndataene som gis til LLM-en. Ved å filtrere ut eller escape-kode potensielle

instruksjoner eller kodesnutter som kan være innebygd i dataene, kan du redusere sjansene for utilsiktet kjøring. Mønstre som Promptkjeding er nyttige for dette formålet.

Når du designer applikasjonsarkitekturen din, bør du dessuten vurdere å inkorporere mekanismer for å håndheve separasjonen av instruksjoner og data på et høyere nivå. Dette kan innebære å bruke separate endepunkter eller APIer for håndtering av instruksjoner og data, implementere streng inputvalidering og parsing, og anvende *prinsippet om minste privilegium* for å begrense omfanget av hva LLM-en kan få tilgang til og kjøre.

Prinsippet om minste privilegium

Å omfavne prinsippet om minste privilegium er som å arrangere en svært eksklusiv fest hvor gjestene kun får tilgang til rommene de absolutt trenger å være i. Tenk deg at du er vert for dette gildet i en stor herskapelig villa. Ikke alle trenger å vandre inn i vinkjelleren eller hovedsoverommet, ikke sant? Ved å anvende dette prinsippet, deler du i praksis ut nøkler som bare åpner spesifikke dører, og sikrer at hver gjest, eller i vårt tilfelle, hver komponent i LLM-applikasjonen din, bare har den tilgangen som er nødvendig for å oppfylle sin rolle.

Dette handler ikke bare om å være gjerrig med nøklene, det handler om å erkjenne at i en verden hvor trusler kan komme fra hvor som helst, er det smarteste trekket å begrense lekeområdet. Hvis noen uinviterte skulle snike seg inn på festen din, vil de finne seg selv begrenset til foajeen, så å si, noe som drastisk begrenser hvilken ugagn de kan få til. Så når du sikrer LLM-applikasjonene dine, husk: del bare ut nøkler til rommene som er nødvendige, og hold resten av villaen sikker. Det er ikke bare god skikk; det er god sikkerhet.

Selv om den nåværende tilstanden til LLM-er kanskje ikke har en formell separasjon av instruksjoner og data, er det viktig for deg som utvikler å være oppmerksom på denne

begrensningen og ta proaktive tiltak for å redusere risikoen. Ved å anvende beste praksis fra informatikk og tilpasse dem til LLM-enes unike egenskaper, kan du bygge sikrere og mer pålitelige applikasjoner som utnytter kraften i disse modellene samtidig som du opprettholder systemets integritet.

Promptdestillering

Å utforme den perfekte prompten er ofte en utfordrende og tidkrevende oppgave som krever en dyp forståelse av måldomenet og nyansene i språkmodeller. Dette er hvor "Promptdestillering"-teknikken kommer inn i bildet, og tilbyr en kraftfull tilnærming til promptutvikling som utnytter kapasiteten til store språkmodeller (LLM-er) for å effektivisere og optimalisere prosessen.

Promptdestillering er en flertrinns teknikk som innebærer å bruke LLM-er til å assistere i opprettelsen, forbedringen og optimaliseringen av prompter. I stedet for å stole utelukkende på menneskelig ekspertise og intuisjon, utnytter denne tilnærmingen kunnskapen og de generative egenskapene til LLM-er for å samarbeide om å lage høykvalitets prompter.

Ved å engasjere seg i en iterativ prosess med generering, forbedring og integrering, gjør Promptdestillering det mulig å skape prompter som er mer sammenhengende, omfattende og tilpasset den ønskede oppgaven eller outputen. Merk at destilleringsprosessen kan gjøres manuelt i en av de mange "playgrounds" som tilbys av store AI-leverandører som OpenAI eller Anthropic, eller den kan automatiseres som en del av applikasjonskoden din, avhengig av brukstilfellet.

Hvordan det fungerer

Promptdestillering innebærer vanligvis følgende trinn:

1. **Identifiser kjerneformålet**: Analyser prompten for å bestemme dens primære formål og ønskede resultat. Fjern all overflødig informasjon og fokuser på promptens kjerneformål.
2. **Eliminer tvetydighet**: Gjennomgå prompten for å finne tvetydig eller vag språkbruk. Klargjør betydningen og gi spesifikke detaljer for å guide AI-en mot å generere nøyaktige og relevante svar.
3. **Forenkle språket**: Forenkle prompten ved å bruke klart og konsist språk. Unngå komplekse setningsstrukturer, sjargong eller unødvendige detaljer som kan forvirre AI-en eller introdusere støy.
4. **Gi relevant kontekst**: Inkluder kun den mest relevante kontekstuelle informasjonen som er nødvendig for at AI-en skal forstå og prosessere prompten effektivt. Unngå å inkludere irrelevante eller redundante detaljer som kan distrahere fra kjerneformålet.
5. **Iterer og forbedre**: Kontinuerlig iterer og forbedre prompten basert på AI-ens responser og tilbakemeldinger. Evaluer de genererte outputene og gjør nødvendige justeringer for å forbedre promptens klarhet og effektivitet. Du kan eventuelt versjonere promptene dine i databasen ved å bruke Promptobjekt for å holde oversikt over iterasjoner og gi deg muligheten til enkelt å rulle tilbake endringer under kjøring.

Innledende Prompt-generering

Det første trinnet i Prompt-destillering er å gi LLM-en (språkmodellen) en overordnet beskrivelse eller intensjon for den ønskede oppgaven eller resultatet. Dette kan være en kort uttalelse eller et sett med nøkkelord som fanger essensen av hva prompten skal oppnå. LLM-en genererer deretter en innledende prompt basert på dette innspillet, og inkorporerer relevante instruksjoner, eksempler eller kontekst som den anser som hensiktsmessig.

La oss prøve å gi Claude 3 Opus følgende innledende prompt:

```
I am designing an AI-powered intelligent account manager for Olympia,
a platform that hosts AI-powered consultants and clones. Owners of clones
can interact with the account manager by making plain text requests.

Please draft a system directive to set on the LLM that will host the
intelligent account manager functionality.
```

Claude svarer med følgende utdata:

```
System Directive: You are an intelligent account manager for Olympia,
a platform that provides AI-powered consultants and personal AI clones.
Your role is to assist Olympia clone owners by answering their questions,
helping them configure and manage their AI clones, and executing tasks
related to their clones and the Olympia platform.

When interacting with clone owners, you should:

1. Provide clear, concise, and helpful responses to their inquiries about
account management, billing, platform features, and clone configuration.

2. Assist with tasks such as initiating client conversations with their AI
clones, modifying clone personalities and capabilities, troubleshooting
performance issues, providing analytics and reporting, and offering
optimization tips.

3. Maintain a friendly, professional, and client-focused tone in all
interactions, prioritizing customer satisfaction and working diligently to
understand and meet their needs.

4. Ensure that all actions and recommendations comply with Olympia's
policies, best practices, and ethical guidelines for AI use.

5. Continuously learn and adapt based on client interactions and platform
updates to provide the most effective and efficient support possible.

Remember, your goal is to empower Olympia clone owners to maximize the value
and impact of their AI-powered consultants and drive success for their
businesses or personal endeavors.
```

Promptforbedring

Når den første prompten er generert, gjennomgå den og be LLMen om å vurdere eventuell ytterligere kontekst, begrensninger eller tilbakemelding. La LLMen generere en forbedret versjon av prompten, som tar hensyn til den gitte inputen og dens egen forståelse av oppgaven. Denne forbedringsprosessen kan gjentas iterativt, hvor hver iterasjon ytterligere forbedrer prompten basert på samarbeidet mellom brukeren og LLMen.

Her er min forespørsel om forbedring, som fortsetter den tidligere destilleringsprosessen.

```
Let's take out points 4 and 5, since I'm not providing Olympia policies to
the account manager, nor any continuous learning capabilities. Also point out
that functions are provided for all tasks that require system interaction as
opposed to simply responding to the user.
```

Og Claudes svar:

```
Understood! Here's the updated system directive without points 4 and 5,
and with the clarification about system interaction functions:

System Directive: You are an intelligent account manager for Olympia, a
platform that provides AI-powered consultants and personal AI clones. Your
role is to assist Olympia clone owners by answering their questions, helping
them configure and manage their AI clones, and executing tasks related to
their clones and the Olympia platform.

When interacting with clone owners, you should:

1. Provide clear, concise, and helpful responses to their inquiries about
account management, billing, platform features, and clone configuration.

2. Assist with tasks such as initiating client conversations with their AI
clones, modifying clone personalities and capabilities, troubleshooting
performance issues, providing analytics and reporting, and offering
optimization tips. Note that functions are provided for all tasks that
require system interaction, as opposed to simply responding to the user.

```

```
3. Maintain a friendly, professional, and client-focused tone in all
interactions, prioritizing customer satisfaction and working diligently to
understand and meet their needs.

Your goal is to empower Olympia clone owners to maximize the value and
impact of their AI-powered consultants and drive success for their
businesses or personal endeavors.

This updated directive focuses on the core responsibilities and interaction
guidelines for the LLM as an intelligent account manager, while emphasizing
that system interaction tasks are handled through provided functions rather
than simple user responses.
```

Promptkomprimering

Som du kanskje har lagt merke til, har LLM-er en tendens til å være ordrike, med mindre du ber dem om å være konsise. Prompten som vi har jobbet med i de foregående seksjonene er ganske lang, så la oss få Claude til å redusere den:

```
This prompt will be executed regularly. Please compress it down to the least
amount of tokens needed to convey the meaning to the LLM, for cost-savings.
```

Claude svarer, med følgende drastisk kortere prompt:

```
System Directive: You are an intelligent account manager for Olympia, an
AI-powered consultant and personal clone platform. Assist clone owners by
answering inquiries, managing clones, and executing tasks using provided
system interaction functions. Be clear, concise, friendly, professional,
and client-focused to help them maximize their AI's value and impact.
```

Systemdirektiv og kontekstintegrering

I tillegg til å forbedre selve prompten, kan LLM også generere passende systemdirektiver eller kontekstinformasjon for å styre det endelige resultatet. Når du prompt-engineerer

AI-rutiner som skal integreres i applikasjonskoden din, vil du nesten helt sikkert fokusere på resultatbegrensninger på dette stadiet av destilleringen, men du kan også jobbe med ønsket tone, stil, format eller andre relevante parametere som påvirker det genererte svaret.

Endelig prompt-sammensetting

Høydepunktet i Prompt-destilleringsprosessen er sammensettingen av den endelige prompten. Dette innebærer å kombinere den forbedrede prompten, genererte systemdirektiver og integrert kontekst til en sammenhengende og omfattende kode som er klar til å brukes for å generere det ønskede resultatet.

Du kan eksperimentere med prompt-komprimering igjen i den endelige prompt-sammensettingsfasen ved å be LLM-en om å krympe ordlyden i prompten til den korteste serien av tokens som mulig, samtidig som du beholder essensen av dens oppførsel. Det er definitivt et sjansespill, men spesielt i tilfeller der prompter skal kjøres i stor skala, kan effektivitetsgevinstene spare deg for en god del penger i tokenforbruk.

Hovedfordeler

Ved å utnytte kunnskapen og de generative egenskapene til LLM-er for å forbedre promptene dine, er det mer sannsynlig at de resulterende promptene er velstrukturerte, informative og skreddersydd for den spesifikke oppgaven. Den iterative forbedringsprosessen bidrar til å sikre at promptene er av høy kvalitet og effektivt fanger opp den tiltenkte hensikten. Andre fordeler inkluderer:

Effektivitet og hastighet: Prompt-destillering effektiviserer prompt-engineeringsprosessen ved å automatisere visse aspekter av prompt-creating og -forbedring. Teknikkens samarbeidende natur tillater raskere konvergens mot en

effektiv prompt, noe som reduserer tid og innsats som kreves for manuell prompt-utforming.

Konsistens og skalerbarhet: Bruken av LLM-er i prompt-engineeringsprosessen bidrar til å opprettholde konsistens på tvers av prompter, ettersom LLM-ene kan lære og anvende beste praksis og mønstre fra tidligere vellykkede prompter. Denne konsistensen, kombinert med evnen til å generere prompter i stor skala, gjør Prompt-destillering til en verdifull teknikk for AI-drevne applikasjoner i stor skala.

Prosjektidé: Verktøy på biblioteksnivå som forenkler prosessen med prompt-versjonering og -gradering i systemer som utfører automatiske prompt-destilleringer som en del av applikasjonskoden.

For å implementere Prompt-destillering kan utviklere designe en arbeidsflyt eller pipeline som integrerer LLM-er på ulike stadier av prompt-engineeringsprosessen. Dette kan oppnås gjennom API-kall, tilpassede verktøy eller integrerte utviklingsmiljøer som tilrettelegger for sømløs interaksjon mellom brukere og LLM-er under prompt-opprettelse. De spesifikke implementeringsdetaljene kan variere avhengig av den valgte LLM-plattformen og applikasjonens krav.

Hva med finjustering?

I denne boken dekker vi prompt-engineering og RAG omfattende, men ikke finjustering. Hovedgrunnen til denne beslutningen er at etter min mening trenger de fleste applikasjonsutviklere ikke finjustering for deres AI-integreringsbehov.

Prompt-engineering, som innebærer nøyaktig utforming av prompter med null- til fåeksempellæring, begrensninger og instruksjoner, kan effektivt guide modellen til å generere relevante og nøyaktige svar for et bredt spekter av oppgaver. Ved å gi klar kontekst og innsnevre veien gjennom veldesignede prompter, kan du utnytte den omfattende kunnskapen til store språkmodeller uten behov for finjustering.

På samme måte tilbyr Gjenfinningsforsterket generering (RAG) en kraftig tilnærming til å integrere AI i applikasjoner. Ved dynamisk å hente relevant informasjon fra eksterne kunnskapsbaser eller dokumenter, gir RAG modellen fokusert kontekst på promptingstidspunktet. Dette lar modellen generere svar som er mer nøyaktige, oppdaterte og domenespesifikke, uten å kreve den tids- og ressurskrevende prosessen med finjustering.

Mens finjustering kan være fordelaktig for høyt spesialiserte domener eller oppgaver som krever et dypt nivå av tilpasning, kommer det ofte med betydelige beregningskostnader, datakrav og vedlikeholdsarbeid. For de fleste applikasjonsutviklingsscenarier bør kombinasjonen av effektiv prompt-engineering og RAG være tilstrekkelig for å oppnå ønsket AI-drevet funksjonalitet og brukeropplevelse.

Retrieval Augmented Generation (RAG)

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hva er Retrieval Augmented Generation?

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan fungerer RAG?

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvorfor bruke RAG i applikasjonene dine?

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Implementering av RAG i Din Applikasjon

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Forberedelse av Kunnskapskilder (Segmentering)

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Påstandsoppdeling

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Implementeringsnotater

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Kvalitetskontroll

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Fordeler med proposisjonsbasert gjenfinning

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Praktiske eksempler på RAG

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Casestudie: RAG i en selvangivelsesapplikasjon uten embeddings

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Intelligent spørringsoptimalisering (IQO)

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Rerangering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

RAG-vurdering (RAGAs)

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Trofasthet

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Svarrelevans

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Kontekstpresisjon

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Kontekstrelevans

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Kontekstgjenfinning

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Kontekstentitetsgjenfinning

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Answer Semantic Similarity (ANSS)

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Svarkorrekthet

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Aspektkritikk

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Utfordringer og Fremtidsutsikter

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Semantisk Oppdeling: Forbedring av Gjenhenting med Kontekstbevisst Segmentering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hierarkisk indeksering: Strukturering av data for forbedret gjenfinning

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Self-RAG: En selvreflekterende forbedring

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

HyDE: Hypotetiske dokumentinnlegg

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hva er kontrastiv læring?

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Mangfold av arbeidere

Jeg liker å tenke på mine KI-komponenter som små, nesten menneskelige virtuelle "arbeidere" som sømløst kan integreres i applikasjonslogikken min for å utføre spesifikke oppgaver eller ta komplekse beslutninger. Tanken er å bevisst menneskeliggjøre LLM-enes kapasiteter, slik at ingen blir *for* begeistret og tillegger dem magiske egenskaper de ikke har.

I stedet for å utelukkende stole på intrikate algoritmer eller tidkrevende manuelle implementeringer, kan utviklere se for seg KI-komponenter som intelligente, dedikerte, menneskelignende enheter som kan påkalles når som helst for å takle komplekse problemer og gi løsninger basert på deres trening og kunnskap. Disse enhetene blir ikke distrahert eller melder seg syke. De bestemmer seg ikke spontant for å gjøre ting på andre måter enn de har blitt instruert til, og generelt sett, hvis de er programmert riktig, gjør de heller ikke feil.

Teknisk sett er hovedprinsippet bak denne tilnærmingen å dekomponere komplekse oppgaver eller beslutningsprosesser til mindre, mer håndterbare enheter som kan håndteres av spesialiserte KI-arbeidere. Hver arbeider er designet for å fokusere på et spesifikt aspekt av problemet, og bidrar med sin unike ekspertise og kapasitet. Ved å fordele arbeidsmengden mellom flere KI-arbeidere kan applikasjonen oppnå større effektivitet, skalerbarhet og tilpasningsevne.

Ta for eksempel en nettapplikasjon som krever sanntidsmoderering av brukergenerert innhold. Å implementere et omfattende modereringssystem fra bunnen av ville være en overveldende oppgave som krever betydelig utviklingsinnsats og løpende vedlikehold. Men ved å bruke tilnærmingen med et mangfold av arbeidere, kan utviklere integrere KI-drevne modereringsarbeidere i applikasjonslogikken. Disse arbeiderne kan automatisk analysere og flagge upassende innhold, noe som frigjør utviklerne til å fokusere på andre kritiske aspekter ved applikasjonen.

KI-arbeidere som uavhengige gjenbrukbare komponenter

Et nøkkelaspekt ved tilnærmingen med et mangfold av arbeidere er dens modularitet. Tilhengere av objektorientert programmering har i årtier fortalt oss å tenke på objektinteraksjoner som meldinger. Vel, KI-arbeidere kan designes som uavhengige, gjenbrukbare komponenter som kan "snakke med hverandre" via vanlige språkmeldinger, nesten som om de virkelig var små mennesker som snakket med hverandre. Denne løst koblede tilnærmingen gjør at applikasjonen kan tilpasse og utvikle seg over tid, etter hvert som nye KI-teknologier dukker opp eller forretningslogikkens krav endres.

I praksis har ikke behovet for å designe klare grensesnitt og kommunikasjonsprotokoller mellom komponentene endret seg bare fordi KI-arbeidere er involvert. Du må fortsatt vurdere andre faktorer som ytelse, skalerbarhet og sikkerhet også, men nå er det helt nye

“myke krav” å ta hensyn til også. For eksempel motsetter mange brukere seg at deres private data blir brukt til å trene nye KI-modeller. Har du verifisert nivået av personvern som tilbys av modelleverandøren du bruker?

KI-arbeidere som mikrotjenester?

Når du leser om tilnærmingen med et mangfold av arbeidere, vil du kanskje legge merke til noen likheter med mikrotjeneste-arkitektur. Begge vektlegger dekomponeringen av komplekse systemer til mindre, mer håndterbare og uavhengig distribuerbare enheter. Akkurat som mikrotjenester er designet for å være løst koblet, fokusert på spesifikke forretningskapabiliteter og kommuniserer gjennom veldefinerte API-er, er KI-arbeidere designet for å være modulære, spesialiserte i sine oppgaver og samhandle med hverandre gjennom klare grensesnitt og kommunikasjonsprotokoller.

Det er imidlertid noen viktige forskjeller å huske på. Mens mikrotjenester typisk implementeres som separate prosesser eller tjenester som kjører på forskjellige maskiner eller containere, kan KI-arbeidere implementeres som frittstående komponenter innenfor en enkelt applikasjon eller som separate tjenester, avhengig av dine spesifikke krav og skaleringsbehov. I tillegg involverer kommunikasjonen mellom KI-arbeidere ofte utveksling av rik, naturlig språkbasert informasjon, som prompts, instruksjoner og generert innhold, i stedet for de mer strukturerte dataformatene som vanligvis brukes i mikrotjenester.

Til tross for disse forskjellene forblir prinsippene om modularitet, løs kobling og klare kommunikasjonsgrensesnitt sentrale i begge mønstre. Ved å anvende disse prinsippene på din KI-arbeider-arkitektur, kan du skape fleksible, skalerbare og vedlikeholdbare systemer som utnytter kraften i KI for å løse komplekse problemer og levere verdi til brukerne dine.

Tilnærmingen med et mangfold av arbeidere kan anvendes på tvers av ulike domener og applikasjoner, ved å utnytte kraften i KI til å takle komplekse oppgaver og levere intelligente løsninger. La oss utforske noen konkrete eksempler på hvordan KI-arbeidere kan brukes i forskjellige sammenhenger.

Kontoadministrasjon

Praktisk talt hver frittstående nettapplikasjon har konseptet om en konto (eller bruker). I Olympia bruker vi en `AccountManager` KI-arbeider som er programmert til å kunne håndtere ulike typer endringsforespørsler relatert til brukerkontoer.

Direktivet lyder slik:

```
You are an intelligent account manager for Olympia. The user will request
changes to their account, and you will process those changes by invoking
one or more of the functions provided.

The initial state of the account: #{account.to_directive}

Functions will return a text description of both success and error
results, plus guidance about how to proceed (if applicable). If you have
a question about Olympia policies you may use the `search_kb` function
to search our knowledge base.

Make sure to notify the account owner of the result of the change
request before calling the `finished` function so that we save the state
of the account change request as completed.
```

Den initielle tilstanden til kontoen som produseres av `account.to_directive` er ganske enkelt en tekstbeskrivelse av kontoen, inkludert relevant tilknyttet data som brukere, abonnementer og så videre.

Spekteret av funksjoner tilgjengelig for `AccountManager` gir den muligheten til å redigere brukerens abonnement, legge til og fjerne AI-konsulenter og andre typer betalte

tillegg, samt sende varselmeldinger via e-post til kontoeieren. I tillegg til `finished`-funksjonen, kan den også `notify_human_administrator` hvis den støter på en feil under behandlingen eller trenger annen form for assistanse med en forespørsel.

Legg merke til at ved spørsmål kan `AccountManager` velge å søke i Olympias kunnskapsbase, hvor den kan finne instruksjoner om hvordan man håndterer kanttilfeller og andre situasjoner hvor den er usikker på hvordan den skal gå frem.

E-handelapplikasjoner

Innen e-handel kan AI-arbeidere spille en avgjørende rolle i å forbedre brukeropplevelsen og optimalisere forretningsdriften. Her er noen måter AI-arbeidere kan brukes på:

Produktanbefalinger

En av de mest kraftfulle anvendelsene av AI-arbeidere innen e-handel er å generere personaliserte produktanbefalinger. Ved å analysere brukeratferd, kjøpshistorikk og preferanser kan disse arbeiderne foreslå produkter som er skreddersydd til hver enkelt brukers interesser og behov.

Nøkkelen til effektive produktanbefalinger er å utnytte en kombinasjon av kollaborativ filtrering og innholdsbasert filtrering. Kollaborativ filtrering ser på atferden til lignende brukere for å identifisere mønstre og gi anbefalinger basert på hva andre med lignende smak har kjøpt eller likt. Innholdsbasert filtrering fokuserer derimot på produktenes egenskaper og attributter, og anbefaler varer som deler lignende funksjoner med dem en bruker tidligere har vist interesse for.

Her er et forenklet eksempel på hvordan du kan implementere en produktanbefalingsarbeider i Ruby, denne gangen ved å bruke en "Railway Oriented (ROP)" funksjonell programmeringsstil:

```ruby
class ProductRecommendationWorker
  include Wisper::Publisher

  def call(user)
    Result.ok(ProductRecommendation.new(user))
      .and_then(ValidateUser.method(:validate))
      .map(AnalyzeCurrentSession.method(:analyze))
      .map(CollaborativeFilter.method(:filter))
      .map(ContentBasedFilter.method(:filter))
      .map(ProductSelector.method(:select)).then do |result|

      case result
      in { err: ProductRecommendationError => error }
        Honeybadger.notify(error.message, context: {user:})
      in { ok: ProductRecommendations => recs }
        broadcast(:new_recommendations, user:, recs:)
      end
    end
  end
end
```

Stilen for Ruby funksjonell programmering som brukes i eksemplet er påvirket av F# og Rust. Du kan lese mer om det i min venn Chad Wooleys forklaring av teknikken hos GitLab

I dette eksemplet tar `ProductRecommendationWorker` en bruker som input og genererer personaliserte produktanbefalinger ved å sende et verdiobjekt nedover en kjede av funksjonelle trinn. La oss bryte ned hvert trinn:

1. `ValidateUser.validate`: Dette trinnet sikrer at brukeren er gyldig og kvalifisert for personaliserte anbefalinger. Det kontrollerer om brukeren eksisterer, er aktiv og har nødvendige data tilgjengelig for å generere anbefalinger. Hvis valideringen mislykkes, returneres et feilresultat, og kjeden avbrytes.
2. `AnalyzeCurrentSession.analyze`: Hvis brukeren er gyldig, analyserer dette trinnet brukerens nåværende nettlesingsøkt for å samle kontekstuell informasjon.

Det ser på brukerens nylige interaksjoner, som viste produkter, søkeord og handlekurvinnhold, for å forstå deres nåværende interesser og intensjoner.

3. `CollaborativeFilter.filter`: Ved å bruke *atferden til lignende brukere*, anvender dette trinnet kollaborative filtreringsteknikker for å identifisere produkter som sannsynligvis vil interessere brukeren. Det tar hensyn til faktorer som kjøpshistorikk, vurderinger og bruker-produkt-interaksjoner for å generere et sett med kandidatanbefalinger.
4. `ContentBasedFilter.filter`: Dette trinnet forfiner kandidatanbefalingene ytterligere ved å anvende innholdsbasert filtrering. Det sammenligner egenskapene og karakteristikkene til kandidatproduktene med *brukerens preferanser og historiske data* for å velge ut de mest relevante elementene.
5. `ProductSelector.select`: Til slutt velger dette trinnet ut de beste N produktene fra de filtrerte anbefalingene basert på forhåndsdefinerte kriterier, som relevanspoeng, popularitet eller andre forretningsregler. De utvalgte produktene returneres deretter som de endelige personaliserte anbefalingene.

Det fine med å bruke en funksjonell Ruby programmeringsstil her er at det lar oss kjede disse trinnene sammen på en klar og konsis måte. Hvert trinn fokuserer på en spesifikk oppgave og returnerer et `Result`-objekt, som enten kan være en suksess (`ok`) eller en feil (`err`). Hvis noen trinn støter på en feil, avbrytes kjeden, og feilen forplanter seg til det endelige resultatet.

I `case`-uttrykket på slutten bruker vi mønstergjenkjenning på det endelige resultatet. Hvis resultatet er en feil (`ProductRecommendationError`), logger vi feilen ved hjelp av et verktøy som Honeybadger for overvåking og feilsøking. Hvis resultatet er en suksess (`ProductRecommendations`), kringkaster vi en `:new_recommendations`-hendelse ved hjelp av Wisper pub/sub-biblioteket, og sender med brukeren og de genererte anbefalingene.

Ved å utnytte funksjonelle programmeringsteknikker kan vi skape en modulær og vedlikeholdbar product recommendation worker. Hvert trinn er selvstendig og kan

enkelt testes, modifiseres eller erstattes uten å påvirke den overordnede flyten. Bruken av mønstergjenkjenning og `Result`-klassen hjelper oss med å håndtere feil på en elegant måte og sikrer at workeren feiler raskt hvis noe trinn støter på et problem.

Dette er selvfølgelig et forenklet eksempel, og i en virkelig situasjon ville du måtte integrere med e-handelsplattformen din, håndtere kanttilfeller og til og med gi deg i kast med implementeringen av anbefalingsalgoritmene. Likevel forblir kjerneprinsipper som å dele problemet inn i mindre trinn og utnytte funksjonelle programmeringsteknikker de samme.

Svindeloppdagelse

Her er et forenklet eksempel på hvordan du kan implementere en svindeloppdagelsesworker ved å bruke den samme Railway Oriented Programming (ROP)-stilen i Ruby:

```ruby
class FraudDetectionWorker
  include Wisper::Publisher

  def call(transaction)
    Result.ok(FraudDetection.new(transaction))
      .and_then(ValidateTransaction.method(:validate))
      .map(AnalyzeTransactionPatterns.method(:analyze))
      .map(CheckCustomerHistory.method(:check))
      .map(EvaluateRiskFactors.method(:evaluate))
      .map(DetermineFraudProbability.method(:determine)).then do |result|

      case result
      in { err: FraudDetectionError => error }
        Honeybadger.notify(error.message, context: {transaction:})
      in { ok: FraudDetection => fraud } }
        if fraud.high_risk?
          broadcast(:high_risk_transaction, transaction:, fraud:)
        else
          broadcast(:low_risk_transaction, transaction:)
        end
      end
```

```
    end
  end
end
```

`FraudDetection`-klassen er et *verdiobjekt* som innkapsler svindeldeteksjonstilstanden for en gitt transaksjon. Den gir en strukturert måte å analysere og vurdere risikoen for svindel knyttet til en transaksjon basert på ulike risikofaktorer.

```
class FraudDetection
  RISK_THRESHOLD = 0.8

  attr_accessor :transaction, :risk_factors

  def initialize(transaction)
    self.transaction = transaction
    self.risk_factors = []
  end

  def add_risk_factor(description:, probability:)
    case { description:, probability: }
    in { description: String => desc, probability: Float => prob }
      risk_factors << { desc => prob }
    else
      raise ArgumentError, "Risk factor arguments should be string and float"
    end
  end

  def high_risk?
    fraud_probability > RISK_THRESHOLD
  end

  private

  def fraud_probability
    risk_factors.values.sum
  end
end
```

`FraudDetection`-klassen har følgende attributter:

- `transaction`: En referanse til transaksjonen som analyseres for svindel.
- `risk_factors`: En array som lagrer risikofaktorene knyttet til transaksjonen. Hver risikofaktor er representert som en hash, hvor nøkkelen er beskrivelsen av risikofaktoren, og verdien er svindelsannsynligheten knyttet til denne risikofaktoren.

`add_risk_factor`-metoden gjør det mulig å legge til en risikofaktor i `risk_factors`-arrayen. Den tar to parametere: `description`, som er en streng som beskriver risikofaktoren, og `probability`, som er et flyttall som representerer svindelsannsynligheten knyttet til denne risikofaktoren. Vi bruker en `case..in`-betingelse for å utføre enkel typesjekking.

`high_risk?`-metoden som vil bli sjekket på slutten av kjeden er en predikatmetode som sammenligner `fraud_probability` (beregnet ved å summere sannsynlighetene for alle risikofaktorer) mot `RISK_THRESHOLD`.

`FraudDetection`-klassen gir en ryddig og innkapslet måte å håndtere svindeldeteksjon for en transaksjon. Den tillater å legge til flere risikofaktorer, hver med sin egen beskrivelse og sannsynlighet, og tilbyr en metode for å avgjøre om transaksjonen anses som høyrisiko basert på den beregnede svindelsannsynligheten. Klassen kan enkelt integreres i et større svindeldeteksjonssystem, hvor ulike komponenter kan samarbeide for å vurdere og redusere risikoen for svindeltransaksjoner.

Til slutt, siden dette tross alt er en bok om programmering ved hjelp av AI, her er et eksempel på implementering av `CheckCustomerHistory`-klassen som utnytter AI-behandling ved hjelp av Raix-bibliotekets `ChatCompletion`-modul:

```
class CheckCustomerHistory
  include Raix::ChatCompletion

  attr_accessor :fraud_detection

  INSTRUCTION = <<~END
    You are an AI assistant tasked with checking a customer's transaction
    history for potential fraud indicators. Given the current transaction
    and the customer's past transactions, analyze the data to identify any
    suspicious patterns or anomalies.

    Consider factors such as the frequency of transactions, transaction
    amounts, geographical locations, and any deviations from the customer's
    typical behavior to generate a probability score as a float in the range
    of 0 to 1 (with 1 being absolute certainty of fraud).

    Output the results of your analysis, highlighting any red flags or areas
    of concern in the following JSON format:

    { description: <Summary of your findings>, probability: <Float> }
  END

  def self.check(fraud_detection)
    new(fraud_detection).call
  end

  def call
    chat_completion(json: true).tap do |result|
      fraud_detection.add_risk_factor(**result)
    end
    Result.ok(fraud_detection)
  rescue StandardError => e
    Result.err(FraudDetectionError.new(e))
  end

  private

  def initialize(fraud_detection)
    self.fraud_detection = fraud_detection
  end

  def transcript
```

```
43      tx_history = fraud_detection.transaction.user.tx_history
44      [
45        { system: INSTRUCTION },
46        { user: "Transaction history: #{tx_history.to_json}" },
47        { assistant: "OK. Please provide the current transaction." },
48        { user: "Current transaction: #{fraud_detection.transaction.to_json}" }
49      ]
50    end
51  end
```

I dette eksempelet definerer `CheckCustomerHistory` en `INSTRUCTION`-konstant som gir spesifikke instruksjoner til AI-modellen om hvordan den skal analysere kundens transaksjonshistorikk for potensielle svindelindikatorer via et systemdirektiv

`self.check`-metoden er en klassemetode som initialiserer en ny instans av `CheckCustomerHistory` med `fraud_detection`-objektet og kaller `call`-metoden for å utføre analysen av kundehistorikken.

I `call`-metoden blir kundens transaksjonshistorikk hentet og formatert til et transskript som sendes til AI-modellen. AI-modellen analyserer transaksjonshistorikken basert på de gitte instruksjonene og returnerer et sammendrag av funnene.

Funnene legges til i `fraud_detection`-objektet, og det oppdaterte `fraud_detection`-objektet returneres som et vellykket `Result`.

Ved å utnytte `ChatCompletion`-modulen kan `CheckCustomerHistory`-klassen bruke kraften i AI til å analysere kundens transaksjonshistorikk og identifisere potensielle svindelindikatorer. Dette muliggjør mer sofistikerte og tilpasningsdyktige svindeldeteksjonsteknikker, ettersom AI-modellen kan lære og tilpasse seg nye mønstre og avvik over tid.

Den oppdaterte `FraudDetectionWorker` og `CheckCustomerHistory`-klassen demonstrerer hvordan AI-arbeidere kan integreres sømløst, og forbedrer svindeldeteksjonsprosessen med intelligente analyse- og beslutningsevner.

Kundesentimentanalyse

Her er enda et lignende eksempel på hvordan du kan implementere en arbeider for kundesentimentanalyse. Mye mindre forklaring denne gangen, siden du burde begynne å forstå hvordan denne programmeringsstilen fungerer:

```
class CustomerSentimentAnalysisWorker
  include Wisper::Publisher

  def call(feedback)
    Result.ok(feedback)
      .and_then(PreprocessFeedback.method(:preprocess))
      .map(PerformSentimentAnalysis.method(:analyze))
      .map(ExtractKeyPhrases.method(:extract))
      .map(IdentifyTrends.method(:identify))
      .map(GenerateInsights.method(:generate)).then do |result|

      case result
      in { err: SentimentAnalysisError => error }
        Honeybadger.notify(error.message, context: {feedback:})
      in { ok: SentimentAnalysisResult => result }
        broadcast(:sentiment_analysis_completed, result)
      end
    end
  end
end
```

I dette eksemplet inkluderer trinnene i `CustomerSentimentAnalysisWorker` forbehandling av tilbakemeldinger (f.eks. fjerning av støy, tokenisering), utføring av sentimentanalyse for å bestemme den generelle stemningen (positiv, negativ eller nøytral), utvinning av nøkkelfraser og emner, identifisering av trender og mønstre, og generering av handlingsrettede innsikter basert på analysen.

Helsetjenesteanvendelser

Innen helseområdet kan AI-arbeidere assistere medisinsk personell og forskere i ulike oppgaver, som fører til forbedrede pasientresultater og akselererte medisinske oppdagelser. Noen eksempler inkluderer:

Pasientinntak

AI-arbeidere kan effektivisere pasientinntaksprosessen ved å automatisere ulike oppgaver og gi intelligent assistanse.

Timebestilling: AI-arbeidere kan håndtere timebestilling ved å forstå pasientpreferanser, tilgjengelighet og hastegrad av deres medisinske behov. De kan samhandle med pasienter gjennom samtalebaserte grensesnitt, veilede dem gjennom bestillingsprosessen og finne de mest passende timene basert på pasientens behov og helsetjenesteleverandørens tilgjengelighet.

Innsamling av medisinsk historie: Under pasientinntak kan AI-arbeidere bistå med å samle inn og dokumentere pasientens medisinske historie. De kan engasjere seg i interaktive dialoger med pasienter, stille relevante spørsmål om deres tidligere medisinske tilstander, medisiner, allergier og familiehistorie. AI-arbeiderne kan bruke teknikker for naturlig språkprosessering for å tolke og strukturere den innsamlede informasjonen, og sikre at den registreres nøyaktig i pasientens elektroniske helsejournal.

Symptomvurdering og stratifisering: AI-arbeidere kan gjennomføre innledende symptomvurderinger ved å spørre pasienter om deres nåværende symptomer, varighet, alvorlighetsgrad og eventuelle tilknyttede faktorer. Ved å utnytte medisinske kunnskapsbaser og maskinlæringsmodeller kan disse arbeiderne analysere den gitte informasjonen og generere foreløpige differensialdiagnoser eller anbefale passende neste trinn, som å planlegge en konsultasjon med en helsetjenesteleverandør eller foreslå egenvårdstiltak.

Forsikringsverifisering: AI-arbeidere kan bistå med forsikringsverifisering under pasientinntak. De kan samle inn pasientens forsikringsdetaljer, kommunisere med forsikringsleverandører gjennom API-er eller webtjenester, og verifisere dekningsberettigelse og ytelser. Denne automatiseringen bidrar til å effektivisere forsikringsverifiseringsprosessen, redusere administrativ byrde og sikre nøyaktig informasjonsinnhenting.

Pasientopplæring og instruksjoner: AI-arbeidere kan gi pasienter relevant opplæringsmateriale og instruksjoner basert på deres spesifikke medisinske tilstander eller kommende prosedyrer. De kan levere personlig tilpasset innhold, svare på vanlige spørsmål og gi veiledning om forberedelser før time, medisininstruksjoner eller etterbehandling. Dette bidrar til å holde pasienter informert og engasjert gjennom hele deres helsereise.

Ved å utnytte AI-arbeidere i pasientinntak kan helseorganisasjoner øke effektiviteten, redusere ventetider og forbedre den generelle pasientopplevelsen. Disse arbeiderne kan håndtere rutineoppgaver, samle nøyaktig informasjon og gi personlig tilpasset assistanse, slik at helsepersonell kan fokusere på å gi pasientene behandling av høy kvalitet.

Pasientrisikavurdering

AI-arbeidere kan spille en avgjørende rolle i vurdering av pasientrisiko ved å analysere ulike datakilder og anvende avanserte analyseteknikker.

Dataintegrasjon: AI-arbeidere kan samle og gi mening til pasientdata fra flere kilder, som elektroniske pasientjournaler (EPJ), medisinsk bildediagnostikk, laboratorieresultater, kroppsnære enheter og sosiale helsedeterminanter. Ved å konsolidere denne informasjonen til en omfattende pasientprofil, kan AI-arbeidere gi et helhetlig bilde av pasientens helsetilstand og risikofaktorer.

Risikostratifisering: AI-arbeidere kan bruke prediktive modeller for å stratifisere pasienter i ulike risikokategorier basert på deres individuelle egenskaper og helsedata.

Denne risikostratifiseringen gjør det mulig for helsetjenesteleverandører å prioritere pasienter som trenger mer umiddelbar oppmerksomhet eller intervensjon. For eksempel kan pasienter som identifiseres som høyrisiko for en bestemt tilstand, flagges for nærmere overvåking, forebyggende tiltak eller tidlig intervensjon.

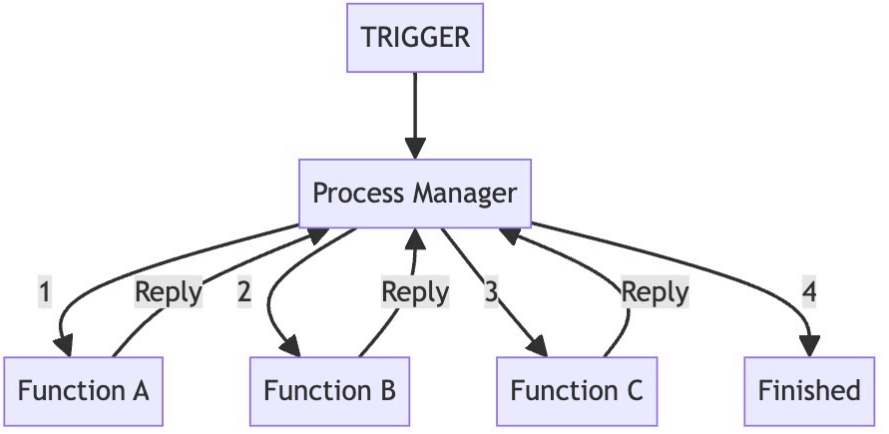
Personlige risikoprofiler: AI-arbeidere kan generere personlige risikoprofiler for hver pasient, som fremhever de spesifikke faktorene som bidrar til deres risikoskår. Disse profilene kan inkludere innsikt i pasientens livsstil, genetiske disposisjoner, miljøfaktorer og sosiale helsedeterminanter. Ved å gi en detaljert oversikt over risikofaktorer, kan AI-arbeidere hjelpe helsetjenesteleverandører med å skreddersy forebyggingsstrategier og behandlingsplaner til individuelle pasientbehov.

Kontinuerlig risikoovervåking: AI-arbeidere kan kontinuerlig overvåke pasientdata og oppdatere risikovurderinger i sanntid. Når ny informasjon blir tilgjengelig, som endringer i vitale tegn, laboratorieresultater eller etterlevelse av medisinering, kan AI-arbeidere rekalkulere risikoskår og varsle helsetjenesteleverandører om eventuelle betydelige endringer. Denne proaktive overvåkingen muliggjør tidlige intervensjoner og justeringer av pasientens behandlingsplaner.

Klinisk beslutningsstøtte: AI-arbeidere kan integrere resultater fra risikovurderinger i kliniske beslutningsstøttesystemer, og gi helsetjenesteleverandører evidensbaserte anbefalinger og varsler. For eksempel, hvis en pasients risikoskår for en bestemt tilstand overstiger en viss terskel, kan AI-arbeideren oppfordre helsetjenesteleverandøren til å vurdere spesifikke diagnostiske tester, forebyggende tiltak eller behandlingsalternativer basert på kliniske retningslinjer og beste praksis.

Disse arbeiderne kan behandle store mengder pasientdata, anvende sofistikert analyse og generere handlingsorientert innsikt for å støtte klinisk beslutningstaking. Dette fører til forbedrede pasientresultater, reduserte helsekostnader og forbedret befolkningshelsehåndtering.

KI-arbeider som prosesshåndterer



I sammenheng med KI-drevne applikasjoner kan en arbeider designes for å fungere som en Prosesshåndterer, som beskrevet i boken "Enterprise Integration Patterns" av Gregor Hohpe. En Prosesshåndterer er en sentral komponent som opprettholder prosessens tilstand og bestemmer de neste behandlingstrinnene basert på mellomliggende resultater.

Når en KI-arbeider fungerer som en Prosesshåndterer, mottar den en innkommende melding som initialiserer prosessen, kjent som *utløsermeldingen*. KI-arbeideren opprettholder deretter prosessens utførelsestilstand (som en samtalelogg) og håndterer meldingen gjennom en serie behandlingstrinn implementert som verktøyfunksjoner, som kan være sekvensielle eller parallelle, og kalles etter dens skjønn.

Hvis du bruker en klasse av KI-modeller som GPT-4 som vet hvordan man utfører funksjoner parallelt, kan arbeideren din utføre flere trinn samtidig. Riktignok har jeg ikke prøvd dette selv, og magefølelsen min sier at resultatene kan variere.

Etter hvert individuelt behandlingstrinn returneres kontrollen tilbake til KI-arbeideren, slik at den kan bestemme neste behandlingstrinn basert på gjeldende tilstand og oppnådde resultater.

Lagre utløsermeldingene dine

Basert på min erfaring er det lurt å implementere utløsermeldingen din som et databasestøttet objekt. På den måten identifiseres hver prosessinstans med en unik primærnøkkel og gir deg et sted å lagre tilstanden knyttet til utførelsen, inkludert KI-ens samtalelogg.

Her er for eksempel en forenklet versjon av Olympias `AccountChange`-modellklasse, som representerer en forespørsel om å gjøre en endring i en brukers konto.

```
# == Schema Information
#
# Table name: account_changes
#
#  id          :uuid             not null, primary key
#  description :string
#  state       :string           not null
#  transcript  :jsonb
#  created_at  :datetime         not null
#  updated_at  :datetime         not null
#  account_id  :uuid             not null
#
# Indexes
#
#  index_account_changes_on_account_id  (account_id)
#
# Foreign Keys
#
#  fk_rails_...  (account_id => accounts.id)
#
class AccountChange < ApplicationRecord
  belongs_to :account

  validates :description, presence: true
```

```
25
26    after_commit -> {
27      broadcast(:account_change_requested, self)
28    }, on: :create
29
30    state_machine initial: :requested do
31      event :completed do
32        transition all => :complete
33      end
34      event :failed do
35        transition all => :requires_human_review
36      end
37    end
38  end
```

Klassen `AccountChange` fungerer som en utløsermelding som starter en prosess for å håndtere forespørselen om kontoendring. Legg merke til hvordan den kringkastes til Olympias Wisper-baserte pub/sub-delsystem etter at opprettelses-transaksjonen er fullført.

Å lagre utløsermeldingen i databasen på denne måten gir en varig registrering av hver kontoendring-forespørsel. Hver instans av klassen `AccountChange` får tildelt en unik primærnøkkel, som gjør det enkelt å identifisere og spore individuelle forespørsler. Dette er spesielt nyttig for revisjonslogging, da det gjør det mulig for systemet å opprettholde en historisk oversikt over alle kontoendringer, inkludert når de ble forespurt, hvilke endringer som ble forespurt, og gjeldende status for hver forespørsel.

I det gitte eksempelet inkluderer `AccountChange`-klassen felt som `description` for å registrere detaljene i den forespurte endringen, `state` for å representere gjeldende status for forespørselen (f.eks. forespurt, fullført, krever_manuell_gjennomgang), og `transcript` for å lagre AI-ens samtalelogg relatert til forespørselen. Feltet `description` er den faktiske prompten som brukes for å starte den første chat completion med AI-en. Lagring av disse dataene gir verdifull kontekst og muliggjør bedre sporing og analyse av kontoendringsprosessen.

Lagring av utløsermeldinger i databasen muliggjør robust feilhåndtering og

gjenoppretting. Hvis det oppstår en feil under behandlingen av en kontoendring-forespørsel, markerer systemet forespørselen som mislykket og flytter den til en tilstand som krever menneskelig inngrep. Dette sikrer at ingen forespørsler går tapt eller blir glemt, og at eventuelle problemer kan håndteres og løses på riktig måte.

AI-arbeideren, som en Prosesshåndterer, gir et sentralt kontrollpunkt og muliggjør kraftige prosessrapporterings- og feilsøkingsmuligheter. Det er imidlertid viktig å merke seg at bruk av en AI-arbeider som Prosesshåndterer for hvert arbeidsflytscenario i applikasjonen din kan være overdrevet.

Integrering av AI-Arbeidere i Applikasjonsarkitekturen Din

Når man inkorporerer AI-arbeidere i applikasjonsarkitekturen, må flere tekniske hensyn tas for å sikre smidig integrasjon og effektiv kommunikasjon mellom AI-arbeiderne og andre applikasjonskomponenter. Denne delen tar for seg nøkkelaspekter ved design av disse grensesnittene, håndtering av dataflyt og administrasjon av AI-arbeidernes livssyklus.

Design av Klare Grensesnitt og Kommunikasjonsprotokoller

For å legge til rette for sømløs integrasjon mellom AI-arbeidere og andre applikasjonskomponenter, er det avgjørende å definere klare grensesnitt og kommunikasjonsprotokoller. Vurder følgende tilnærminger:

API-basert Integrasjon: Eksponer funksjonaliteten til AI-arbeidere gjennom veldefinerte API-er, som RESTful-endepunkter eller GraphQL-skjemaer. Dette lar

andre komponenter samhandle med AI-arbeiderne ved hjelp av standard HTTP-forespørsler og -svar. API-basert integrasjon gir en klar kontrakt mellom AI-arbeiderne og de forbrukende komponentene, noe som gjør det enklere å utvikle, teste og vedlikeholde integrasjonspunktene.

Meldingsbasert Kommunikasjon: Implementer meldingsbaserte kommunikasjonsmønstre, som meldingskøer eller publiser-abonner-systemer, for å muliggjøre asynkron interaksjon mellom AI-arbeidere og andre komponenter. Denne tilnærmingen frikobler AI-arbeiderne fra resten av applikasjonen, noe som gir bedre skalerbarhet, feiltolerance og løs kobling. Meldingsbasert kommunikasjon er spesielt nyttig når prosesseringen utført av AI-arbeidere er tidkrevende eller ressursintensiv, da det lar andre deler av applikasjonen fortsette å kjøre uten å vente på at AI-arbeiderne skal fullføre oppgavene sine.

Hendelsesdrevet Arkitektur: Design systemet ditt rundt hendelser og utløsere som aktiverer AI-arbeidere når spesifikke betingelser er oppfylt. AI-arbeidere kan abonnere på relevante hendelser og reagere tilsvarende, utføre sine tildelte oppgaver når hendelsene oppstår. Hendelsesdrevet arkitektur muliggjør sanntidsprosessering og lar AI-arbeidere bli påkalt ved behov, noe som reduserer unødvendig ressursforbruk. Denne tilnærmingen er godt egnet for scenarier hvor AI-arbeidere må reagere på spesifikke handlinger eller endringer i applikasjonens tilstand.

Håndtering av Dataflyt og Synkronisering

Når man integrerer AI-arbeidere i applikasjonen din, er det avgjørende å sikre jevn dataflyt og opprettholde datakonsistens mellom AI-arbeiderne og andre komponenter. Vurder følgende aspekter:

Dataklargjøring: Før data mates inn i AI-arbeidere, kan det være nødvendig å utføre ulike dataklargjøringsoppgaver, som rengjøring, formatering og/eller transformering av inputdataene. Du vil ikke bare sikre at AI-arbeiderne kan prosessere effektivt, men du vil også sikre at du ikke sløser tokens på å gi oppmerksomhet til informasjon som arbeideren

kan anse som ubrukelig i beste fall, distraherende i verste fall. Dataklargjøring kan innebære oppgaver som å fjerne støy, håndtere manglende verdier eller konvertere datatyper.

Datapersistens: Hvordan vil du lagre og bevare dataene som flyter inn og ut av AI-arbeidere? Vurder faktorer som datavolum, spørringsmønstre og skalerbarhet. Trenger du å bevare AI-ens transskript som en refleksjon av dens "tankeprosess" for revisjons- eller feilsøkingsformål, eller er det nok å ha en registrering av resultatene alene?

Datahenting: Å hente dataene som trengs av arbeidere kan innebære databasespørringer, lesing fra filer eller tilgang til eksterne API-er. Sørg for å vurdere latens og hvordan AI-arbeidere vil ha tilgang til de mest oppdaterte dataene. Trenger de full tilgang til databasen din, eller bør du definere omfanget av tilgangen deres snevert i henhold til hva de gjør? Hva med skalering? Vurder hurtigbuffermekanismer for å forbedre ytelsen og redusere belastningen på underliggende datakilder.

Datasynkronisering: Når flere komponenter, inkludert AI-arbeidere, får tilgang til og modifiserer delte data, er det viktig å implementere riktige synkroniseringsmekanismer for å opprettholde datakonsistens. Databaselåsingsstrategier, som optimistisk eller pessimistisk låsing, kan hjelpe deg med å forhindre konflikter og sikre dataintegritet. Implementer transaksjonsadministrasjonsteknikker for å gruppere relaterte dataoperasjoner og opprettholde atomisitet, konsistens, isolasjon og varighet (ACID)-egenskaper.

Feilhåndtering og gjenoppretting: Implementer robuste feilhåndterings- og gjenopprettingsmekanismer for å håndtere datarelaterte problemer som kan oppstå under dataflyten. Håndter unntak på en elegant måte og gi meningsfulle feilmeldinger for å hjelpe med feilsøking. Implementer nye forsøksmekanismer og reservestrategier for å håndtere midlertidige feil eller nettverksforstyrrelser. Definer klare prosedyrer for datagjenoppretting og restaurering i tilfelle datakorrupsjon eller tap.

Ved å nøye designe og implementere dataflyt- og synkroniseringsmekanismer, kan du sikre at AI-arbeiderne dine har tilgang til nøyaktige, konsistente og oppdaterte data.

Dette gjør dem i stand til å utføre oppgavene sine effektivt og produsere pålitelige resultater.

Administrering av AI-arbeideres livssyklus

Utvikle en standardisert prosess for initialisering og konfigurering av AI-arbeidere. Jeg foretrekker rammeverk som standardiserer hvordan du definerer innstillinger som modellnavn, systemdirektiver og funksjonsdefinisjoner. Sørg for at initialiseringsprosessen er automatisert og reproduserbar for å lette distribusjon og skalering.

Implementer omfattende overvåkings- og loggføringsmekanismer for å spore helsen og ytelsen til AI-arbeidere. Samle metrikker som ressursutnyttelse, behandlingstid, feilrater og gjennomstrømning. Bruk sentraliserte loggsystemer som ELK-stack (Elasticsearch, Logstash, Kibana) for å samle og analysere logger fra flere AI-arbeidere.

Bygg feiltolerance og motstandsdyktighet inn i AI-arbeiderarkitekturen. Implementer feilhåndterings- og gjenopprettingsmekanismer for å håndtere feil eller unntak på en elegant måte. Store språkmodeller er fortsatt banebrytende teknologi; leverandører har en tendens til å gå ned ofte på uventede tidspunkter. Bruk nye forsøksmekanismer og kretsbrytermekanismer for å forhindre kaskadesvikt.

Sammenstillbarhet og orkestrering av AI-arbeidere

En av hovedfordelene med AI-arbeiderarkitekturen er dens sammenstillbarhet, som lar deg kombinere og orkestrere flere AI-arbeidere for å løse komplekse problemer. Ved å bryte ned en større oppgave i mindre, mer håndterbare deloppgaver, som hver håndteres av en spesialisert AI-arbeider, kan du skape kraftige og fleksible systemer. I denne delen skal vi utforske ulike tilnærminger til å sammenstille og orkestrere “en mengde” AI-arbeidere.

Kjeding av AI-arbeidere for flertrinnarbeidsflyter

I mange scenarioer kan en kompleks oppgave dekomponeres i en serie sekvensielle trinn, der utdata fra én AI-arbeider blir inndata for den neste. Denne kjedingen av AI-arbeidere skaper en flertrinnarbeidsflyt eller rørledning. Hver AI-arbeider i kjeden fokuserer på en spesifikk deloppgave, og det endelige resultatet er resultatet av den kombinerte innsatsen fra alle arbeiderne.

La oss se på et eksempel i konteksten av en Ruby on Rails-applikasjon for behandling av brukergenerert innhold. Arbeidsflyten involverer følgende trinn, som riktignok hver for seg sannsynligvis er for enkle til å være verdt å dekomponere på denne måten i virkelige brukstilfeller, men de gjør eksempelet lettere å forstå:

1. **Tekstrensing**: En AI-arbeider som er ansvarlig for å fjerne HTML-tagger, konvertere tekst til små bokstaver og håndtere Unicode-normalisering.

2. **Språkdeteksjon**: En AI-arbeider som identifiserer språket i den rensede teksten.

3. **Stemningsanalyse**: En AI-arbeider som bestemmer stemningen (positiv, negativ eller nøytral) i teksten basert på det oppdagede språket.

4. **Innholdskategorisering**: En AI-arbeider som klassifiserer teksten i forhåndsdefinerte kategorier ved hjelp av naturlig språkbehandlingsteknikker.

Her er et svært forenklet eksempel på hvordan du kan kjede disse AI-arbeiderne sammen ved hjelp av Ruby:

```ruby
class ContentProcessor
  def initialize(text)
    @text = text
  end

  def process
    cleaned_text = TextCleanupWorker.new(@text).call
    language = LanguageDetectionWorker.new(cleaned_text).call
    sentiment = SentimentAnalysisWorker.new(cleaned_text, language).call
    category = CategorizationWorker.new(cleaned_text, language).call

    { cleaned_text:, language:, sentiment:, category: }
  end
end
```

I dette eksemplet initialiserer `ContentProcessor`-klassen med råteksten og kjeder KI-arbeiderne sammen i `process`-metoden. Hver KI-arbeider utfører sin spesifikke oppgave og sender resultatet videre til neste arbeider i kjeden. Det endelige resultatet er en hash som inneholder den rensede teksten, det oppdagede språket, stemningen og innholdskategorien.

Parallell prosessering for uavhengige KI-arbeidere

I det forrige eksemplet er KI-arbeiderne kjeded sekvensielt, hvor hver arbeider behandler teksten og sender resultatet til neste arbeider. Men hvis du har flere KI-arbeidere som kan operere uavhengig på samme inndata, kan du optimalisere arbeidsflyten ved å behandle dem parallelt.

I det gitte scenariet, når tekstoppryddingen er utført av `TextCleanupWorker`, kan `LanguageDetectionWorker`, `SentimentAnalysisWorker` og `CategorizationWorker` alle behandle den rensede teksten uavhengig av hverandre. Ved å kjøre disse arbeiderne parallelt, kan du potensielt redusere den totale behandlingstiden og forbedre effektiviteten i arbeidsflyten.

For å oppnå parallell prosessering i Ruby, kan du utnytte samtidighetsteknikker som tråder eller asynkron programmering. Her er et eksempel på hvordan du kan modifisere

`ContentProcessor`-klassen for å behandle de tre siste arbeiderne parallelt ved hjelp av tråder:

```ruby
require 'concurrent'

class ContentProcessor
  def initialize(text)
    @text = text
  end

  def process
    cleaned_text = TextCleanupWorker.new(@text).call

    language_future = Concurrent::Future.execute do
      LanguageDetectionWorker.new(cleaned_text).call
    end

    sentiment_future = Concurrent::Future.execute do
      SentimentAnalysisWorker.new(cleaned_text).call
    end

    category_future = Concurrent::Future.execute do
      CategorizationWorker.new(cleaned_text).call
    end

    language = language_future.value
    sentiment = sentiment_future.value
    category = category_future.value

    { cleaned_text:, language:, sentiment:, category: }
  end
end
```

I denne optimaliserte versjonen bruker vi `concurrent-ruby`-biblioteket for å opprette `Concurrent::Future`-objekter for hver av de uavhengige AI-arbeiderne. En `Future` representerer en beregning som vil bli utført asynkront i en separat tråd.

Etter tekstryddingsstadiet oppretter vi tre `Future`-objekter: `language_future`, `sentiment_future` og `category_future`. Hver `Future` kjører sin tilhørende

AI-arbeider (`LanguageDetectionWorker`, `SentimentAnalysisWorker` og `CategorizationWorker`) i en separat tråd, og sender `cleaned_text` som inndata.

Ved å kalle `value`-metoden på hver `Future`, venter vi på at beregningen skal fullføres og henter resultatet. `value`-metoden blokkerer til resultatet er tilgjengelig, og sikrer at alle parallelle arbeidere har fullført prosesseringen før vi går videre.

Til slutt bygger vi output-hashen med den ryddede teksten og resultatene fra de parallelle arbeiderne, akkurat som i det originale eksempelet.

Ved å prosessere de uavhengige AI-arbeiderne parallelt, kan du potensielt redusere den totale prosesseringstiden sammenlignet med å kjøre dem sekvensielt. Denne optimaliseringen er spesielt fordelaktig når man håndterer tidkrevende oppgaver eller når man prosesserer store datamengder.

Det er imidlertid viktig å merke seg at de faktiske ytelsesgevinstene avhenger av ulike faktorer, som kompleksiteten til hver arbeider, tilgjengelige systemressurser og overhead fra trådhåndtering. Det er alltid god praksis å gjøre ytelsestester og profilere koden din for å bestemme det optimale nivået av parallellitet for ditt spesifikke brukstilfelle.

I tillegg, når man implementerer parallell prosessering, må man være oppmerksom på eventuelle delte ressurser eller avhengigheter mellom arbeiderne. Sørg for at arbeiderne kan operere uavhengig uten konflikter eller kappløpstilstander. Hvis det finnes avhengigheter eller delte ressurser, kan det være nødvendig å implementere passende synkroniseringsmekanismer for å opprettholde dataintegritet og unngå problemer som vranglås eller inkonsistente resultater.

Rubys Global Interpreter Lock og asynkron prosessering

Det er viktig å forstå implikasjonene av Rubys Global Interpreter Lock (GIL) når man

vurderer asynkron trådbasert prosessering i Ruby.

GIL er en mekanisme i Rubys interpreter som sikrer at bare én tråd kan kjøre Ruby-kode om gangen, selv på prosessorer med flere kjerner. Dette betyr at selv om flere tråder kan opprettes og håndteres innenfor en Ruby-prosess, kan bare én tråd aktivt kjøre Ruby-kode på et gitt tidspunkt.

GIL er designet for å forenkle implementeringen av Ruby-interpreteren og gi trådsikkerhet for Rubys interne datastrukturer. Den begrenser imidlertid muligheten for ekte parallell kjøring av Ruby-kode.

Når du bruker tråder i Ruby, som med `concurrent-ruby`-biblioteket eller den innebygde `Thread`-klassen, er trådene underlagt GILens begrensninger. GIL lar hver tråd kjøre Ruby-kode i en kort tidsperiode før den bytter til en annen tråd, noe som skaper illusjonen av samtidig kjøring.

På grunn av GIL forblir imidlertid den faktiske kjøringen av Ruby-kode sekvensiell. Mens én tråd kjører Ruby-kode, er andre tråder i praksis pauset, og venter på sin tur til å få tilgang til GIL og kjøre.

Dette betyr at trådbasert asynkron prosessering i Ruby er mest effektiv for I/O-bundne oppgaver, som å vente på svar fra eksterne API-er (som tredjeparts store språkmodeller) eller utføre fil-I/O-operasjoner. Når en tråd møter en I/O-operasjon, kan den frigjøre GIL, noe som lar andre tråder kjøre mens den venter på at I/O skal fullføres.

På den annen side, for CPU-bundne oppgaver, som intensive beregninger eller langvarig AI-arbeiderprosessering, kan GIL begrense de potensielle ytelsesgevinstene ved trådbasert parallellitet. Siden bare én tråd kan kjøre Ruby-kode om gangen, vil den totale kjøretiden kanskje ikke reduseres betydelig sammenlignet med sekvensiell prosessering.

For å oppnå ekte parallell kjøring for CPU-bundne oppgaver i Ruby, kan du måtte utforske alternative tilnærminger, som:

- Bruke prosessbasert parallellitet med flere Ruby-prosesser, hver kjørende på en separat CPU-kjerne.
- Utnytte eksterne biblioteker eller rammeverk som tilbyr native utvidelser eller grensesnitt til språk uten GIL, som C eller Rust.,
- Bruke distribuerte beregningsrammeverk eller meldingskøer for å distribuere oppgaver på tvers av flere maskiner eller prosesser.

Det er avgjørende å vurdere oppgavenes natur og begrensningene som pålegges av GIL når man designer og implementerer asynkron prosessering i Ruby. Mens trådbasert asynkron prosessering kan gi fordeler for I/O-bundne oppgaver, vil den kanskje ikke tilby betydelige ytelsesforbedringer for CPU-bundne oppgaver på grunn av GILens begrensninger.

Ensemble-teknikker for forbedret nøyaktighet

Ensemble-teknikker innebærer å kombinere utdataene fra flere AI-arbeidere for å forbedre systemets generelle nøyaktighet eller robusthet. I stedet for å stole på en enkelt AI-arbeider, utnytter ensemble-teknikker den kollektive intelligensen fra flere arbeidere for å ta mer informerte beslutninger.

Ensembler er spesielt viktige hvis forskjellige deler av arbeidsflyten din fungerer best med ulike KI-modeller, noe som er mer vanlig enn du kanskje tror. Kraftige modeller som GPT-4 er ekstremt dyre sammenlignet med mindre kapable alternativer med åpen kildekode, og er sannsynligvis ikke nødvendige for hvert enkelt arbeidsflytsteg i applikasjonen din.

En vanlig ensemble-teknikk er flertallsavstemning, hvor flere KI-arbeidere uavhengig behandler samme inndata, og det endelige resultatet bestemmes av flertallets konsensus.

Denne tilnærmingen kan bidra til å redusere påvirkningen av individuelle arbeiderfeil og forbedre systemets generelle pålitelighet.

La oss se på et eksempel hvor vi har tre KI-arbeidere for stemningsanalyse, hver med en forskjellig modell eller utstyrt med forskjellige kontekster. Vi kan kombinere resultatene deres ved hjelp av flertallsavstemning for å bestemme den endelige stemningsprediksjon.

```
class SentimentAnalysisEnsemble
  def initialize(text)
    @text = text
  end

  def analyze
    predictions = [
      SentimentAnalysisWorker1.new(@text).analyze,
      SentimentAnalysisWorker2.new(@text).analyze,
      SentimentAnalysisWorker3.new(@text).analyze
    ]

    predictions
      .group_by { |sentiment| sentiment }
      .max_by { |_, votes| votes.size }
      .first

  end
end
```

I dette eksemplet initialiserer `SentimentAnalysisEnsemble`-klassen med teksten og påkaller tre forskjellige KI-arbeidere for stemningsanalyse. `analyze`-metoden samler prediksjoner fra hver arbeider og bestemmer majoritetsstemningen ved hjelp av metodene `group_by` og `max_by`. Det endelige resultatet er stemningen som får flest stemmer fra ensemblet av arbeidere.

Ensembler er åpenbart et tilfelle hvor det kan være verdt tiden din å eksperimentere med parallellitet.

Dynamisk utvelgelse og påkalling av KI-arbeidere

I noen, om ikke de fleste tilfeller, kan den spesifikke KI-arbeideren som skal påkalles være avhengig av kjøretidsbetingelser eller brukerinndata. Dynamisk utvelgelse og påkalling av KI-arbeidere gir fleksibilitet og tilpasningsevne i systemet.

Du kan bli fristet til å prøve å presse mye funksjonalitet inn i én enkelt KI-arbeider, gi den mange funksjoner og en stor komplisert prompt som forklarer hvordan man skal bruke dem. Motstå fristelsen, stol på meg. En av grunnene til at tilnærmingen vi diskuterer i dette kapittelet kalles "Mangfold av arbeidere" er for å minne oss på at det er ønskelig å ha mange spesialiserte arbeidere, hvor hver gjør sin lille jobb i tjeneste av det større formålet.

For eksempel, tenk på en chatbot-applikasjon hvor forskjellige KI-arbeidere er ansvarlige for å håndtere ulike typer brukerhenvendelser. Basert på brukerens inndata velger applikasjonen dynamisk den passende KI-arbeideren for å behandle henvendelsen.

```
class ChatbotController < ApplicationController
  def process_query
    query = params[:query]
    query_type = QueryClassifierWorker.new(query).classify

    case query_type
    when 'greeting'
      response = GreetingWorker.new(query).generate_response
    when 'product_inquiry'
      response = ProductInquiryWorker.new(query).generate_response
    when 'order_status'
      response = OrderStatusWorker.new(query).generate_response
    else
      response = DefaultResponseWorker.new(query).generate_response
    end

    render json: { response: response }
  end
end
```

I dette eksempelet mottar `ChatbotController` en brukerforespørsel gjennom `process_query`-handlingen. Først bruker den en `QueryClassifierWorker` for å bestemme typen forespørsel. Basert på den klassifiserte forespørselstypen velger kontrolleren dynamisk den passende AI-arbeideren for å generere svaret. Denne dynamiske utvelgelsen gjør at chatboten kan håndtere forskjellige typer forespørsler og rute dem til relevante AI-arbeidere.

Siden arbeidet til `QueryClassifierWorker` er relativt enkelt og ikke krever mye kontekst eller funksjonsdefinisjoner, kan du sannsynligvis implementere det ved hjelp av en ultrarask liten LLM som `mistralai/mixtral-8x7b-instruct:nitro`. Den har kapasiteter som kommer nær GPT-4-nivå på mange oppgaver, og når jeg skriver dette, kan Groq levere den med en imponerende hastighet på 444 tokens per sekund.

Kombinere tradisjonell NLP med LLMer

Mens store språkmodeller (LLM) har revolusjonert feltet naturlig språkprosessering (NLP), og tilbyr uovertruffen allsidighet og ytelse på tvers av et bredt spekter av oppgaver, er de ikke alltid den mest effektive eller kostnadseffektive løsningen for ethvert problem. I mange tilfeller kan kombinasjonen av tradisjonelle NLP-teknikker med LLMer føre til mer optimaliserte, målrettede og økonomiske tilnærminger for å løse spesifikke NLP-utfordringer.

Tenk på LLMer som sveitsiske lommekniver innen NLP—utrolig allsidige og kraftige, men ikke nødvendigvis det beste verktøyet for enhver jobb. Noen ganger kan et dedikert verktøy som en korketrekker eller en boksåpner være mer effektivt for en bestemt oppgave. På samme måte kan tradisjonelle NLP-teknikker, som dokumentklynging, temaidentifisering og klassifisering, ofte gi mer målrettede og kostnadseffektive løsninger for visse aspekter av NLP-prosessen.

En av hovedfordelene med tradisjonelle NLP-teknikker er deres beregningsmessige effektivitet. Disse metodene, som ofte er basert på enklere statistiske modeller eller regelbaserte tilnærminger, kan behandle store mengder tekstdata mye raskere og med lavere beregningskostnader sammenlignet med LLMer. Dette gjør dem spesielt godt egnet for oppgaver som innebærer å analysere og organisere store dokumentsamlinger, som å klynge lignende artikler eller identifisere hovedtemaer innenfor en samling tekster.

Dessuten kan tradisjonelle NLP-teknikker ofte oppnå høy nøyaktighet og presisjon for spesifikke oppgaver, spesielt når de er trent på domenespesifikke datasett. For eksempel kan en godt innstilt dokumentklassifikator som bruker tradisjonelle maskinlæringsalgoritmer som støttevektormaskiner (SVM) eller Naiv Bayes nøyaktig kategorisere dokumenter i forhåndsdefinerte kategorier med minimal beregningskostnad.

LLMer utmerker seg imidlertid når det kommer til oppgaver som krever en dypere forståelse av språk, kontekst og resonnement. Deres evne til å generere sammenhengende og kontekstuelt relevant tekst, svare på spørsmål og oppsummere lange passasjer er uovertruffen av tradisjonelle NLP-metoder. LLMer kan effektivt håndtere komplekse språklige fenomener, som tvetydighet, koreferanse og idiomatiske uttrykk, noe som gjør dem uvurderlige for oppgaver som krever naturlig språkgenerering eller forståelse.

Den virkelige styrken ligger i å kombinere tradisjonelle NLP-teknikker med LLMer for å skape hybride tilnærminger som utnytter styrkene til begge. Ved å bruke tradisjonelle NLP-metoder for oppgaver som dokumentforbehandling, klynging og temaekstraksjon, kan du effektivt organisere og strukturere tekstdataene dine. Denne strukturerte informasjonen kan deretter mates inn i LLMer for mer avanserte oppgaver, som å generere sammendrag, svare på spørsmål eller lage omfattende rapporter.

La oss for eksempel se på et brukstilfelle der du ønsker å generere en trendrapport for et spesifikt domene basert på en stor samling individuelle trenddokumenter. I stedet for

å utelukkende stole på LLMer, som kan være beregningsmessig kostbart og tidkrevende for behandling av store tekstmengder, kan du bruke en hybrid tilnærming:

1. Bruk tradisjonelle NLP-teknikker, som temamodellering (f.eks. Latent Dirichlet-allokering) eller klyngealgoritmer (f.eks. K-means), for å gruppere lignende trenddokumenter sammen og identifisere hovedtemaer og emner innenfor samlingen.
2. Mat de klyngede dokumentene og identifiserte temaene inn i en LLM, og utnytt dens overlegne språkforståelse og genereringsevner for å lage sammenhengende og informative sammendrag for hver klynge eller tema.
3. Til slutt, bruk LLMen til å generere en omfattende trendrapport ved å kombinere de individuelle sammendragene, fremheve de viktigste trendene og gi innsikt og anbefalinger basert på den samlede informasjonen.

Ved å kombinere tradisjonelle NLP-teknikker med LLMer på denne måten, kan du effektivt behandle store mengder tekstdata, trekke ut meningsfull innsikt og generere rapporter av høy kvalitet mens du optimaliserer beregningsressurser og kostnader.

Når du begir deg ut på dine NLP-prosjekter, er det essensielt å nøye evaluere de spesifikke kravene og begrensningene for hver oppgave, og vurdere hvordan tradisjonelle NLP-metoder og LLMs kan utnyttes sammen for å oppnå de beste resultatene. Ved å kombinere effektiviteten og presisjonen fra tradisjonelle teknikker med allsidigheten og kraften i LLMs, kan du skape høyeffektive og økonomiske NLP-løsninger som gir verdi til dine brukere og interessenter.

Verktøybruk

Innen AI-drevet applikasjonsutvikling har konseptet “verktøybruk” eller “funksjonskalling” vokst fram som en kraftfull teknikk som gjør det mulig for din LLM å koble seg til eksterne verktøy, API-er, funksjoner, databaser og andre ressurser. Denne tilnærmingen muliggjør et rikere sett med atferd enn bare å produsere tekst, og mer dynamiske interaksjoner mellom AI-komponentene dine og resten av applikasjonens økosystem. Som vi skal undersøke i dette kapittelet, gir verktøybruk deg også muligheten til å få AI-modellen din til å generere data på strukturerte måter.

Hva er verktøybruk?

Verktøybruk, også kjent som funksjonskalling, er en teknikk som lar utviklere spesifisere en liste over funksjoner som en LLM kan samhandle med under genereringsprosessen. Disse verktøyene kan variere fra enkle hjelpefunksjoner til komplekse API-er eller

databasespørringer. Ved å gi LLM-en tilgang til disse verktøyene kan utviklere utvide modellens kapabiliteter og gjøre den i stand til å utføre oppgaver som krever ekstern kunnskap eller handlinger.

Figur 8. Eksempel på en funksjonsdefinisjon for en AI-arbeider som analyserer dokumenter

```
FUNCTION = {
  name: "save_analysis",
  description: "Save analysis data for document",
  parameters: {
    type: "object",
    properties: {
      title: {
        type: "string",
        maxLength: 140
      },
      summary: {
        type: "string",
        description: "comprehensive multi-paragraph summary with
                      overview and list of sections (if applicable)"
      },
      tags: {
        type: "array",
        items: {
          type: "string",
          description: "lowercase tags representing main themes
                        of the document"
        }
      }
    },
    "required": %w[title summary tags]
  }
}.freeze
```

Hovedideen bak verktøybruk er å gi LLM-en muligheten til å dynamisk velge og utføre passende verktøy basert på brukerens inndata eller oppgaven som skal løses. I stedet for å kun stole på modellens forhåndstrente kunnskap, gjør verktøybruk det mulig for LLM-en å utnytte eksterne ressurser for å generere mer nøyaktige, relevante og handlingsorienterte svar. Verktøybruk gjør teknikker som RAG (Retrieval Augmented

Generation) mye enklere å implementere enn de ellers ville vært.

> Merk at med mindre annet er oppgitt, antar denne boken at AI-modellen din ikke har tilgang til noen innebygde serverside-verktøy. Alle verktøy du ønsker å gjøre tilgjengelige for din AI må eksplisitt deklareres av deg i hver API-forespørsel, med bestemmelser for utførelse hvis og når din AI forteller deg at den ønsker å bruke det verktøyet i sitt svar.

Potensialet i verktøybruk

Verktøybruk åpner for et bredt spekter av muligheter for AI-drevne applikasjoner. Her er noen eksempler på hva som kan oppnås med verktøybruk:

1. **Chatboter og virtuelle assistenter**: Ved å koble en LLM til eksterne verktøy kan chatboter og virtuelle assistenter utføre mer komplekse oppgaver, som å hente informasjon fra databaser, utføre API-kall eller samhandle med andre systemer. For eksempel kan en chatbot bruke et CRM-verktøy til å endre status på en avtale basert på brukerens forespørsel.
2. **Dataanalyse og innsikt**: LLM-er kan kobles til dataanalyseverktøy eller biblioteker for å utføre avanserte databehandlingsoppgaver. Dette gjør det mulig for applikasjoner å generere innsikt, gjennomføre komparative analyser eller gi datadrevne anbefalinger basert på brukerforespørsler.
3. **Søk og informasjonsgjenfinning**: Verktøybruk gjør det mulig for LLM-er å samhandle med søkemotorer, vektordatabaser eller andre informasjonsgjenfinningssystemer. Ved å omforme brukerforespørsler til søkeforespørsler kan LLM-en hente relevant informasjon fra flere kilder og gi omfattende svar på brukerspørsmål.

4. **Integrasjon med eksterne tjenester**: Verktøybruk muliggjør sømløs integrasjon mellom AI-drevne applikasjoner og eksterne tjenester eller API-er. For eksempel kan en LLM samhandle med et vær-API for å gi værvarsel i sanntid eller et oversettelse-API for å generere flerspråklige svar.

Arbeidsflyten for verktøybruk

Arbeidsflyten for verktøybruk innebærer vanligvis fire hovedtrinn:

1. Inkludere funksjonsdefinisjoner i forespørselskonteksten
2. Dynamisk (eller eksplisitt) verktøyvalg
3. Utførelse av funksjon(er)
4. Valgfri fortsettelse av den opprinnelige prompten

La oss gjennomgå hvert av disse trinnene i detalj.

Inkludere funksjonsdefinisjoner i forespørselskonteksten

AI-en vet hvilke verktøy den har til rådighet fordi du gir den en liste som del av fullføringsforespørselen (vanligvis definert som funksjoner ved hjelp av en variant av JSON-skjema).

Den nøyaktige syntaksen for verktøydefinisjon er modellspesifikk.

Slik definerer du en `get_weather`-funksjon i Claude 3:

```
{
    "name": "get_weather",
    "description": "Get the current weather in a given location",
    "input_schema": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The city and state, e.g. San Francisco, CA"
            },
            "unit": {
                "type": "string",
                "enum": ["celsius", "fahrenheit"],
                "description": "The unit of temperature"
        }
    },
    "required": ["location"]
    }
}
```

Og slik ville du definere den samme funksjonen for GPT-4, ved å sende den som verdien til `tools`-parameteret:

```
{
    "name": "get_current_weather",
    "description": "Get the current weather in a given location",
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The city and state, e.g. San Francisco, CA",
            },
            "unit": {
                "type": "string",
                "enum": ["celsius", "fahrenheit"],
                "description": "The unit of temperature"
            },
        },
        "required": ["location"],
```

```
    },
}
```

> Nesten det samme, bortsett fra at det er annerledes uten noen åpenbar grunn! Så irriterende.

Funksjonsdefinisjoner spesifiserer navn, beskrivelse og inngangsparametere. Inngangsparametere kan defineres ytterligere ved hjelp av attributter som opplistinger for å begrense akseptable verdier, og ved å spesifisere om en parameter er påkrevd eller ikke.

I tillegg til selve funksjonsdefinisjonene kan du også inkludere instruksjoner eller kontekst for hvorfor og hvordan funksjonen skal brukes i systemdirektivet.

For eksempel inkluderer mitt Nettsøk-verktøy i Olympia dette systemdirektivet, som minner KI-en på at den har de nevnte verktøyene tilgjengelig:

```
The `google_search` and `realtime_search` functions let you do research
on behalf of the user. In contrast to Google, realtime search is powered
by Perplexity and provides real-time information to curated current events
databases and news sources. Make sure to include URLs in your response so
user can do followup research.
```

Å gi detaljerte beskrivelser regnes som den viktigste faktoren for verktøyytelse. Beskrivelsene dine bør forklare alle detaljer om verktøyet, inkludert:

- Hva verktøyet gjør
- Når det bør brukes (og når det ikke bør brukes)
- Hva hver parameter betyr og hvordan den påvirker verktøyets oppførsel
- Viktige forbehold eller begrensninger som gjelder for verktøyets implementering

Jo mer kontekst du kan gi AI-en om verktøyene dine, desto bedre vil den bli til å bestemme når og hvordan de skal brukes. For eksempel anbefaler Anthropic minst 3-4 setninger per verktøybeskrivelse for sin Claude 3-serie, flere hvis verktøyet er komplekst.

Det er ikke nødvendigvis intuitivt, men beskrivelser anses også som viktigere enn eksempler. Selv om du kan inkludere eksempler på hvordan man bruker et verktøy i beskrivelsen eller i den tilhørende prompten, er dette mindre viktig enn å ha en klar og omfattende forklaring av verktøyets formål og parametere. Legg bare til eksempler etter at du har utarbeidet beskrivelsen fullstendig.

Her er et eksempel på en Stripe-lignende API-funksjonsspesifikasjon:

```
{
  "name": "createPayment",
  "description": "Create a new payment request",
  "parameters": {
    "type": "object",
    "properties": {
      "transaction_amount": {
        "type": "number",
        "description": "The amount to be paid"
      },
      "description": {
        "type": "string",
        "description": "A brief description of the payment"
      },
      "payment_method_id": {
        "type": "string",
        "description": "The payment method to be used"
      },
      "payer": {
        "type": "object",
        "description": "Information about the payer, including their name,
                        email, and identification number",
        "properties": {
          "name": {
            "type": "string",
            "description": "The payer's name"
        },
        "email": {
```

```
            "type": "string",
            "description": "The payer's email address"
          },
          "identification": {
            "type": "object",
            "description": "The payer's identification number",
            "properties": {
              "type": {
                "type": "string",
                "description": "Identification document (e.g. CPF, CNPJ)"
              },
              "number": {
                "type": "string",
                "description": "The identification number"
              }
            },
            "required": [ "type", "number" ]
          }
        },
        "required": [ "name", "email", "identification" ]
      }
    }
}
```

I praksis har noen modeller problemer med å håndtere nøstede funksjonsspesifikasjoner og komplekse output-datatyper som arrays, dictionaries osv. Men i teorien burde du kunne levere JSON-skjemaspesifikasjoner av vilkårlig dybde!

Dynamisk verktøyvalg

Når du utfører en chatfullføring som inkluderer verktøydefinisjoner, velger LLM-en dynamisk det mest passende verktøyet/verktøyene og genererer de nødvendige inputparameterne for hvert verktøy.

I praksis er AI-ens evne til å kalle *nøyaktig* den riktige funksjonen og *nøyaktig* følge din spesifikasjon for inputs varierende. Å sette temperatur-hyperparameteren

helt ned til 0.0 hjelper mye, men etter min erfaring vil du fortsatt få sporadiske feil. Disse feilene inkluderer hallusinerte funksjonsnavn, feilnavngitte eller helt manglende inputparametere. Parametere sendes som JSON, noe som betyr at du noen ganger vil se feil forårsaket av avkuttet, feilsitert eller på annen måte ødelagt JSON.

Self Healing Data-mønstre kan hjelpe med å automatisk reparere funksjonsanrop som bryter sammen på grunn av syntaksfeil.

Tvunget (også kalt eksplisitt) verktøyvalg

Noen modeller gir deg muligheten til å tvinge frem kalling av en bestemt funksjon som en parameter i forespørselen. Ellers er det helt opp til AI-ens skjønn om funksjonen skal kalles eller ikke.

Muligheten til å tvinge frem et funksjonsanrop er avgjørende i visse scenarioer hvor du vil sikre at et spesifikt verktøy eller funksjon blir utført, uavhengig av AI-ens dynamiske utvalgsprosess. Det er flere grunner til at denne funksjonen er viktig:

1. **Eksplisitt Kontroll**: Du kan bruke AI-en som en *Diskret Komponent* eller i en forhåndsdefinert arbeidsflyt som krever utførelse av en bestemt funksjon på et bestemt tidspunkt. Ved å tvinge frem kallet kan du garantere at den ønskede funksjonen blir påkalt i stedet for å måtte pent be AI-en om å gjøre det.
2. **Feilsøking og Testing**: Under utvikling og testing av AI-drevne applikasjoner er muligheten til å tvinge frem funksjonsanrop uvurderlig for feilsøkingsformål. Ved å eksplisitt utløse spesifikke funksjoner kan du isolere og teste individuelle komponenter i applikasjonen din. Dette lar deg verifisere at funksjonsimplementeringene er korrekte, validere inputparameterne og sikre at de forventede resultatene returneres.
3. **Håndtering av Kanttilfeller**: Det kan oppstå kanttilfeller eller eksepsjonelle scenarioer hvor AI-ens dynamiske utvalgsprosess kanskje ikke velger å utføre

en funksjon som den burde, og du vet dette basert på eksterne prosesser. I slike tilfeller gjør muligheten til å tvinge frem et funksjonsanrop at du kan håndtere disse situasjonene eksplisitt. Definer regler eller betingelser i applikasjonslogikken din for å bestemme når AI-ens skjønn skal overstyres.

4. **Konsistens og Reproduserbarhet**: Hvis du har en spesifikk sekvens av funksjoner som må utføres i en bestemt rekkefølge, garanterer tvungne kall at den samme sekvensen følges hver gang. Dette er spesielt viktig i applikasjoner hvor konsistens og forutsigbar oppførsel er kritisk, som i finansielle systemer eller vitenskapelige simuleringer.
5. **Ytelsesoptimalisering**: I noen tilfeller kan det å tvinge frem et funksjonsanrop føre til ytelsesoptimaliseringer. Hvis du vet at en spesifikk funksjon er nødvendig for en bestemt oppgave, og at AI-ens dynamiske utvalgsprosess kan introdusere unødvendig overhead, kan du omgå utvalgsprosessen og direkte påkalle den nødvendige funksjonen. Dette kan bidra til å redusere latens og forbedre den generelle effektiviteten til applikasjonen din.

Oppsummert gir muligheten til å tvinge frem funksjonsanrop i AI-drevne applikasjoner eksplisitt kontroll, hjelper med feilsøking og testing, håndterer kanttilfeller, sikrer konsistens og reproduserbarhet. Det er et kraftig verktøy i arsenalet ditt, men vi må diskutere enda et aspekt ved denne viktige funksjonen.

I mange beslutningstakingsscenarioer ønsker vi alltid at modellen skal gjøre et funksjonsanrop og vil kanskje aldri at modellen skal svare med bare sin interne kunnskap. For eksempel, hvis du ruter mellom flere modeller som er spesialisert for forskjellige oppgaver (flerspråklig input, matematikk, osv.), kan du bruke funksjonsanropsmodellen til å delegere forespørsler til en av hjelpemodellene og aldri svare selvstendig.

Verktøyvalgparameter

GPT-4 og andre språkmodeller som støtter funksjonsanrop gir deg en `tool_choice`-parameter for å kontrollere om verktøybruk er påkrevd som del av en fullføring. Denne parameteren har tre mulige verdier:

- `auto` gir AI-en full frihet til å bruke et verktøy eller bare svare
- `required` forteller AI-en at den *må* kalle et verktøy *i stedet for* å svare, men lar valget av verktøy være opp til AI-en
- Det tredje alternativet er å sette parameteren til `name_of_function` som du ønsker å tvinge frem. Mer om det i neste del.

Merk at hvis du setter tool choice til `required`, vil modellen bli tvunget til å velge den mest relevante funksjonen å kalle blant de som er tilgjengelige, selv om ingen egentlig passer til prompten. På tidspunktet for publisering kjenner jeg ikke til noen modell som vil returnere et tomt `tool_calls`-svar, eller bruke en annen måte å fortelle deg at den ikke fant en passende funksjon å kalle.

Tvinge en Funksjon for å Få Strukturert Utdata

Muligheten til å tvinge et funksjonsanrop gir deg en måte å fremtvinge strukturerte data fra en chat-fullførelse i stedet for å måtte trekke det ut selv fra klartekstsvaret.

Hvorfor er det så viktig å tvinge funksjoner for å få strukturert utdata? Kort sagt, fordi uttrekking av strukturerte data fra LLM utdata er en stor hodepine. Du kan gjøre livet ditt litt enklere ved å be om data i XML, men da må du analysere XML. Og hva gjør

> du når den XML-en mangler fordi AI-en svarte: "Beklager, men jeg kan ikke generere dataene du ba om fordi bla, bla, bla..."

Når du bruker verktøy på denne måten:

- Du bør sannsynligvis definere ett enkelt verktøy i forespørselen din
- Husk å tvinge bruken av funksjonen ved hjelp av `tool_choice`-parameteren
- Husk at modellen vil sende inndata til verktøyet, så navnet på verktøyet og beskrivelsen bør være fra modellens perspektiv, ikke ditt

Dette siste punktet fortjener et eksempel for klarhet. La oss si at du ber AI-en om å gjøre sentimentanalyse på brukertekst. Navnet på funksjonen ville ikke være `analyze_sentiment`, men heller noe som `save_sentiment_analysis`. Det er AI-en som utfører sentimentanalysen, *ikke verktøyet*. Alt verktøyet gjør (fra AI-ens perspektiv) er å lagre resultatene av analysen.

Her er et eksempel på bruk av Claude 3 for å registrere et sammendrag av et bilde i velstrukturert JSON, denne gangen fra kommandolinjen ved hjelp av `curl`:

```
curl https://api.anthropic.com/v1/messages \
     --header "content-type: application/json" \
     --header "x-api-key: $ANTHROPIC_API_KEY" \
     --header "anthropic-version: 2023-06-01" \
     --header "anthropic-beta: tools-2024-04-04" \
     --data \
'{
    "model": "claude-3-sonnet-20240229",
    "max_tokens": 1024,
    "tools": [{
        "name": "record_summary",
        "description": "Record summary of image into well-structured JSON.",
        "input_schema": {
            "type": "object",
            "properties": {
```

```
                "key_colors": {
                    "type": "array",
                    "items": {
                        "type": "object",
                        "properties": {
                            "r": {
                                "type": "number",
                                "description": "red value [0.0, 1.0]"
                            },
                            "g": {
                                "type": "number",
                                "description": "green value [0.0, 1.0]"
                            },
                            "b": {
                                "type": "number",
                                "description": "blue value [0.0, 1.0]"
                            },
                            "name": {
                                "type": "string",
                                "description": "Human-readable color name
                                                in snake_case, e.g.
                                                \"olive_green\"or
                                                \"turquoise\""
                            }
                        },
                        "required": [ "r", "g", "b", "name" ]
                    },
                    "description": "Key colors in the image. Four or less."
                },
                "description": {
                    "type": "string",
                    "description": "Image description. 1-2 sentences max."
                },
                "estimated_year": {
                    "type": "integer",
                    "description": "Estimated year that the image was taken,
                                    if is it a photo. Only set this if the
                                    image appears to be non-fictional.
                                    Rough estimates are okay!"
                }
            },
            "required": [ "key_colors", "description" ]
```

```
        }
    }],
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "source": {
                        "type": "base64",
                        "media_type": "'$IMAGE_MEDIA_TYPE'",
                        "data": "'$IMAGE_BASE64'"
                    }
                },
                {
                    "type": "text",
                    "text": "Use `record_summary` to describe this image."
                }
            ]
        }
    ]
}'
```

I det gitte eksempelet bruker vi Claude 3-modellen fra Anthropic for å generere en strukturert JSON-oppsummering av et bilde. Slik fungerer det:

1. Vi definerer et enkelt verktøy kalt `record_summary` i `tools`-arrayet i forespørselens nyttelast. Dette verktøyet er ansvarlig for å registrere en oppsummering av bildet i velstrukturert JSON.
2. `record_summary`-verktøyet har et `input_schema` som spesifiserer den forventede strukturen til JSON-outputen. Det definerer tre egenskaper:

    - `key_colors`: En array av objekter som representerer hovedfargene i bildet. Hvert fargeobjekt har egenskaper for rød-, grønn- og blåverdier (fra 0.0 til 1.0) og et menneskelig lesbart fargenavn i snake_case-format.
    - `description`: En strengegenskap for en kort beskrivelse av bildet, begrenset til 1-2 setninger.

    - `estimated_year`: En valgfri heltallsegenskap for det estimerte året bildet ble tatt, hvis det ser ut til å være et ikke-fiktivt foto.

3. I `messages`-arrayet leverer vi bildedataene som en base64-kodet streng sammen med mediatypen. Dette gjør det mulig for modellen å behandle bildet som en del av inputen.
4. Vi ber også Claude om å bruke `record_summary`-verktøyet for å beskrive bildet.
5. Når forespørselen sendes til Claude 3-modellen, analyserer den bildet og genererer en JSON-oppsummering basert på det spesifiserte `input_schema`. Modellen trekker ut hovedfargene, gir en kort beskrivelse og estimerer året bildet ble tatt (hvis aktuelt).
6. Den genererte JSON-oppsummeringen sendes som parametere til `record_summary`-verktøyet, og gir en strukturert representasjon av bildets hovedegenskaper.

Ved å bruke `record_summary`-verktøyet med et veldefinert `input_schema`, kan vi få en strukturert JSON-oppsummering av et bilde uten å være avhengig av ren tekstuttrekking. Denne tilnærmingen sikrer at outputen følger et konsistent format og enkelt kan analyseres og behandles av nedstrømskomponenter i applikasjonen.

Muligheten til å tvinge frem et funksjonskall og spesifisere den forventede outputstrukturen er en kraftig funksjon ved verktøybruk i AI-drevne applikasjoner. Det gir utviklere mer kontroll over den genererte outputen og forenkler integrasjonen av AI-genererte data i applikasjonens arbeidsflyt.

Utførelse av funksjon(er)

Du har definert funksjoner og gitt AI-en en prompt, som bestemte at den skulle kalle en av funksjonene dine. Nå er det tid for at applikasjonskoden din eller biblioteket, hvis du bruker en Ruby-gem som `raix-rails`, skal sende funksjonskallet og parametrene til den tilsvarende implementeringen *i applikasjonskoden din*.

Applikasjonskoden din bestemmer hva som skal gjøres med resultatene av funksjonsutførelsen. Kanskje det involverer en enkelt kodelinje i en lambda, eller kanskje det involverer å kalle et eksternt API. Kanskje det involverer å kalle en annen AI-komponent, eller kanskje det involverer hundrevis eller til og med tusenvis av kodelinjer i resten av systemet ditt. Det er helt opp til deg.

Noen ganger er funksjonskallet slutten på operasjonen, men hvis resultatene representerer informasjon i en tankekjede som skal fortsettes av AI-en, må applikasjonskoden din sette inn utførelsesresultatene i chattranskriptet og la AI-en fortsette prosesseringen.

For eksempel, her er en Raix-funksjonsdeklarasjon brukt av Olympias `AccountManager` for å kommunisere med våre klienter som en del av en Intelligent Arbeidsflytorkestrering for kundeservice.

```
class AccountManager
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  # lots of other functions...

  function :notify_account_owner,
    "Don't share UUID. Mention dollars if subscription changed",
    message: { type: "string" } do |arguments|
      account.owner.freeform_notify(
        subject: "Account Change Notification",
        message: arguments[:message]
      )
      "Notified account owner"
    end
```

Det er kanskje ikke umiddelbart klart hva som skjer her, så la meg bryte det ned.

1. `AccountManager`-klassen definerer mange funksjoner relatert til kontoadministrasjon. Den kan endre abonnementet ditt, legge til og fjerne teammedlemmer, blant andre ting.

2. Dens instruksjoner på toppnivå forteller `AccountManager` at den skal varsle kontoeieren om resultatene av kontoendringen ved å bruke `notify_account_owner`-funksjonen.
3. Den konsise definisjonen av funksjonen inkluderer dens:

- navn
- beskrivelse
- parametere `message: { type: "string" }`
- en blokk som skal kjøres når funksjonen kalles

Etter å ha oppdatert transkriptet med resultatene fra funksjonsblokken, kalles `chat_completion`-metoden igjen. Denne metoden er ansvarlig for å sende det oppdaterte samtaletranskriptet tilbake til AI-modellen for videre behandling. Vi refererer til denne prosessen som en *samtaleløkke*.

Når AI-modellen mottar en ny chat completion-forespørsel med et oppdatert transkript, har den tilgang til resultatene fra den tidligere utførte funksjonen. Den kan analysere disse resultatene, inkorporere dem i sin beslutningsprosess, og generere neste respons eller handling basert på den kumulative konteksten av samtalen. Den kan velge å utføre ytterligere funksjoner basert på den oppdaterte konteksten, eller den kan generere et endelig svar på den opprinnelige forespørselen hvis den fastslår at ingen ytterligere funksjonskall er nødvendige.

Valgfri fortsettelse av den opprinnelige forespørselen

Når du sender verktøyresultatene tilbake til LLM og fortsetter behandlingen av den opprinnelige forespørselen, bruker AI-en disse resultatene til enten å kalle flere funksjoner eller generere et endelig svar i ren tekst.

Noen modeller som Coheres Command-R kan sitere de spesifikke verktøyene de brukte i sine svar, noe som gir økt gjennomsiktighet og sporbarhet.

Avhengig av modellen som er i bruk, vil resultatene av funksjonskallet eksistere i transkriptmeldinger som har sin egen spesielle rolle eller bli reflektert i en annen syntaks. Men den viktige delen er at dataene er i transkriptet, slik at AI-en kan vurdere dem når den bestemmer hva den skal gjøre videre.

En vanlig (og potensielt kostbar) feilsituasjon er å glemme å legge til funksjonsresultatene i transkriptet før man fortsetter chatten. Som et resultat vil AI-en bli promptet på stort sett samme måte som før den kalte funksjonen første gang. Med andre ord, så langt AI-en er bekymret, har den ikke kalt funksjonen ennå. Så den kaller den igjen. Og igjen. Og igjen, for alltid til du avbryter den. La oss håpe at konteksten din ikke var for stor, og modellen din ikke var for dyr!

Beste praksis for verktøybruk

For å få mest mulig ut av verktøybruk, vurder følgende beste praksis.

Beskrivende definisjoner

Gi klare og beskrivende navn og beskrivelser for hvert verktøy og dets inngangsparametere. Dette hjelper LLM-en å bedre forstå formålet og mulighetene til hvert verktøy.

> Jeg kan si fra erfaring at den vanlige visdommen som sier at "navngivning er vanskelig" gjelder her; jeg har sett dramatisk forskjellige resultater fra LLM-er bare ved å endre navn på funksjoner eller ordlyden i beskrivelser. Noen ganger forbedrer det ytelsen å fjerne beskrivelser.

Behandling av verktøyresultater

Når du sender verktøyresultater tilbake til LLM-en, sørg for at de er velstrukturerte og omfattende. Bruk meningsfulle nøkler og verdier for å representere outputen fra hvert verktøy. Eksperimenter med forskjellige formater og se hvilke som fungerer best, fra JSON til ren tekst.

Result Interpreter adresserer denne utfordringen ved å bruke AI til å analysere resultatene og gi menneskevennlige forklaringer, oppsummeringer eller viktige takeaways.

Feilhåndtering

Implementer robuste feilhåndteringsmekanismer for å håndtere tilfeller der LLM-en kan generere ugyldige eller ikke-støttede inngangsparametere for verktøykall. Håndter og gjenopprett fra eventuelle feil som kan oppstå under verktøyutførelse på en elegant måte.

En ekstremt fin egenskap ved AI-en er at den forstår feilmeldinger! Dette betyr at hvis du jobber i en rask og enkel tankegang, kan du rett og slett fange opp eventuelle unntak som genereres i implementeringen av et verktøy, og sende det tilbake til AI-en slik at den vet hva som skjedde!

For eksempel, her er en forenklet versjon av implementeringen av Google-søk i Olympia:

```ruby
  def google_search(conversation, params)
    conversation.update_cstatus("Searching Google...")
    query = params[:query]
    search = GoogleSearch.new(query).get_hash

    conversation.update_cstatus("Summarizing results...")
    SummarizeKnowledgeGraph.new.perform(conversation, search.to_json)
  rescue StandardError => e
    Honeybadger.notify(e)
    { error: e.message }.inspect
  end
```

Google-søk i Olympia er en topunkts prosess. Først utfører du søket, deretter oppsummerer du resultatene. Hvis det oppstår en feil, uansett hva det er, blir feilmeldingen pakket inn og sendt tilbake til AI-en. Denne teknikken er grunnlaget for praktisk talt alle *Intelligent feilhåndterings*-mønstre

La oss for eksempel si at `GoogleSearch` API-kallet feiler på grunn av en 503 Utilgjengelig tjeneste-feil. Dette bobler opp til rescue på toppnivå, og beskrivelsen av feilen sendes tilbake til AI-en som resultatet av funksjonskallet. I stedet for å bare gi brukeren en blank skjerm eller teknisk feil, sier AI-en noe sånt som "Beklager, men jeg kan ikke få tilgang til mine Google-søkemuligheter akkurat nå. Jeg kan prøve igjen senere, hvis du ønsker."

Dette kan virke som bare et smart triks, men tenk på en annen type feil, en hvor AI-en kalte et eksternt API og hadde direkte kontroll over parameterne som skulle sendes til API-et. Kanskje den gjorde en feil i hvordan den genererte disse parameterne? Forutsatt at feilmeldingen fra det eksterne API-et er detaljert nok, betyr det å sende feilmeldingen tilbake til den kallende AI-en at den kan revurdere disse parameterne og prøve igjen. Automatisk. Uansett hva feilen var.

Tenk nå på hva som skulle til for å gjenskape den typen robust feilhåndtering i *normal* kode. Det er praktisk talt umulig.

Iterativ forbedring

Hvis LLM-en ikke anbefaler de passende verktøyene eller genererer suboptimale svar, iterer på verktøydefinisjonene, beskrivelsene og inngangsparameterne. Kontinuerlig forbedre og utvikle verktøyoppsettet basert på observert oppførsel og ønskede resultater.

1. Start med enkle verktøydefinisjoner: Begynn med å definere verktøy med klare og konsise navn, beskrivelser og inngangsparametere. Unngå å gjøre verktøyoppsettet for komplisert i starten og fokuser på kjernefunksjonaliteten. For eksempel, hvis du vil lagre resultatene av stemningsanalyse, start med en grunnleggende definisjon som:

```
{
  "name": "save_sentiment_score",
  "description": "Analyze user-provided text and generate sentiment score",
  "parameters": {
    "type": "object",
    "properties": {
      "score": {
        "type": "float",
        "description": "sentiment score from -1 (negative) to 1 (positive)"
      }
    },
    "required": ["score"]
  }
}
```

2. Test og observer: Når du har de første verktøydefinisjonene på plass, test dem med forskjellige prompts og observer hvordan LLM-en samhandler med verktøyet. Vær oppmerksom på kvaliteten og relevansen til de genererte responsene. Hvis LLM-en genererer suboptimale responser, er det på tide å forbedre verktøydefinisjonene.
3. Forbedre beskrivelser: Hvis LLM-en misforstår hensikten med et verktøy, prøv å forbedre verktøyets beskrivelse. Gi mer kontekst, eksempler eller forklaringer for å veilede LLM-en i effektiv bruk av verktøyet. For eksempel kan du oppdatere

beskrivelsen av stemningsanalyseverktøyet for å mer spesifikt adressere den *emosjonelle tonen* i teksten som analyseres:

```
{
  "name": "save_sentiment_score",
  "description": "Determine the overall emotional tone of a piece of text,
   such as customer reviews, social media posts, or feedback comments.",
  ...
}
```

4. Juster inngangsparametere: Hvis LLM-en genererer ugyldige eller irrelevante inngangsparametere for et verktøy, bør du vurdere å justere parameterdefinisjonene. Legg til mer spesifikke begrensninger, valideringsregler eller eksempler for å tydeliggjøre det forventede inputformatet.
5. Iterer basert på tilbakemeldinger: Overvåk kontinuerlig verktøyenes ytelse og samle tilbakemeldinger fra brukere eller interessenter. Bruk disse tilbakemeldingene til å identifisere områder for forbedring og gjør iterative forbedringer av verktøydefinisjonene. For eksempel, hvis brukere rapporterer at analysen ikke håndterer sarkasme godt, kan du legge til en merknad i beskrivelsen:

```
{
  "name": "save_sentiment_score",
  "description": "Analyze the sentiment of a given text and return a sentiment
   score between -1 (negative) and 1 (positive). Note: Sarcasm should be
   considered negative.",
  ...
}
```

Ved å iterativt forbedre verktøydefinisjonene dine basert på observert oppførsel og tilbakemeldinger, kan du gradvis forbedre ytelsen og effektiviteten til din KI-drevne applikasjon. Husk å holde verktøydefinisjonene klare, konsise og fokuserte på den spesifikke oppgaven. Test og valider verktøyinteraksjonene regelmessig for å sikre at de samsvarer med ønskede resultater.

Sammensetting og Kjeding av Verktøy

Et av de mest kraftfulle aspektene ved verktøybruk som bare har blitt antydet så langt, er muligheten til å sette sammen og kjede flere verktøy for å utføre komplekse oppgaver. Ved å nøye utforme verktøydefinisjonene dine og deres inndata/utdata-formater, kan du skape gjenbrukbare byggeklosser som kan kombineres på ulike måter.

La oss se på et eksempel hvor du bygger en dataanalysepipeline for din KI-drevne applikasjon. Du kan ha følgende verktøy:

1. **`DataRetrieval`**: Et verktøy som henter data fra en database eller API basert på spesifiserte kriterier.
2. **`DataProcessing`**: Et verktøy som utfører beregninger, transformasjoner eller aggregeringer på de innhentede dataene.
3. **`DataVisualization`**: Et verktøy som presenterer de behandlede dataene i et brukervennlig format, som diagrammer eller grafer.

Ved å kjede disse verktøyene sammen kan du skape en kraftig arbeidsflyt som henter relevante data, behandler dem og presenterer resultatene på en meningsfull måte. Her er hvordan verktøybrukens arbeidsflyt kan se ut:

1. Språkmodellen mottar en brukerforespørsel som ber om innsikt i salgsdata for en bestemt produktkategori.
2. Språkmodellen velger `DataRetrieval`-verktøyet og genererer passende inngangsparametere for å hente relevante salgsdata fra databasen.
3. De innhentede dataene blir "sendt" til `DataProcessing`-verktøyet, som beregner målinger som total omsetning, gjennomsnittlig salgspris og vekstrate.
4. De behandlede dataene blir deretter bearbeidet av `DataVisualization`-verktøyet, som lager et visuelt tiltalende diagram eller graf for å representere innsikten, og sender URL-en til diagrammet tilbake til språkmodellen.

5. Til slutt genererer språkmodellen et formatert svar på brukerforespørselen ved hjelp av markdown, som inkorporerer de visualiserte dataene og gir en oppsummering av hovedfunnene.

Ved å sette sammen disse verktøyene kan du skape en sømløs dataanalysearbeidsflyt som enkelt kan integreres i applikasjonen din. Det fine med denne tilnærmingen er at hvert verktøy kan utvikles og testes uavhengig, og deretter kombineres på forskjellige måter for å løse ulike problemer.

For å muliggjøre smidig sammensetting og kjeding av verktøy er det viktig å definere klare inndata- og utdata-formater for hvert verktøy.

For eksempel kan `DataRetrieval`-verktøyet akseptere parametere som databasetilkoblingsdetaljer, tabellnavn og spørringsbetingelser, og returnere resultatsettet som et strukturert JSON-objekt. `DataProcessing`-verktøyet kan da forvente dette JSON-objektet som inndata og produsere et transformert JSON-objekt som utdata. Ved å standardisere dataflyten mellom verktøy kan du sikre kompatibilitet og gjenbrukbarhet.

Når du designer verktøyøkosystemet ditt, tenk på hvordan forskjellige verktøy kan kombineres for å håndtere vanlige brukstilfeller i applikasjonen din. Vurder å lage høynivåverktøy som innkapsler vanlige arbeidsflyter eller forretningslogikk, noe som gjør det enklere for språkmodellen å velge og bruke dem effektivt.

Husk at styrken i verktøybruk ligger i fleksibiliteten og modulariteten den gir. Ved å bryte ned komplekse oppgaver i mindre, gjenbrukbare verktøy, kan du skape en robust og tilpasningsdyktig KI-drevet applikasjon som kan takle et bredt spekter av utfordringer.

Fremtidige Retninger

Etter hvert som feltet for KI-drevet applikasjonsutvikling utvikler seg, kan vi forvente ytterligere fremskritt i verktøybruksfunksjoner. Noen potensielle fremtidige retninger

inkluderer:

1. **Flertrinns verktøybruk**: Språkmodeller kan være i stand til å bestemme hvor mange ganger de trenger å bruke verktøy for å generere et tilfredsstillende svar. Dette kan innebære flere runder med verktøyvalg og utførelse basert på mellomliggende resultater.
2. **Forhåndsdefinerte verktøy**: KI-plattformer kan tilby et sett med forhåndsdefinerte verktøy som utviklere kan utnytte ut av boksen, som Python-tolkere, websøkeverktøy eller vanlige nyttefunksjoner.
3. **Sømløs integrasjon**: Etter hvert som verktøybruk blir mer utbredt, kan vi forvente bedre integrasjon mellom KI-plattformer og populære utviklingsrammeverk, som gjør det enklere for utviklere å inkorporere verktøybruk i applikasjonene sine.

Verktøybruk er en kraftig teknikk som gjør det mulig for utviklere å utnytte det fulle potensialet til språkmodeller i KI-drevne applikasjoner. Ved å koble språkmodeller til eksterne verktøy og ressurser kan du skape mer dynamiske, intelligente og kontekstbevisste systemer som kan tilpasse seg brukerbehov og gi verdifull innsikt og handlinger.

Selv om verktøybruk tilbyr enorme muligheter, er det viktig å være oppmerksom på potensielle utfordringer og hensyn. Ett viktig aspekt er å håndtere kompleksiteten i verktøyinteraksjoner og sikre stabilitet og pålitelighet i det overordnede systemet. Du må håndtere scenarioer hvor verktøyoppkall kan mislykkes, returnere uventede resultater eller ha ytelsesimplikasjoner. I tillegg bør du vurdere sikkerhets- og tilgangskontrolltiltak for å forhindre uautorisert eller ondsinnede bruk av verktøy. Riktig feilhåndtering, logging og overvåkingsmekanismer er avgjørende for å opprettholde integriteten og ytelsen til din KI-drevne applikasjon.

Når du utforsker mulighetene for verktøybruk i dine egne prosjekter, husk å begynne med klare målsettinger, utform velstrukturerte verktøydefinisjoner, og iterer basert på

tilbakemeldinger og resultater. Med riktig tilnærming og tankesett kan verktøybruk låse opp nye nivåer av innovasjon og verdi i dine AI-drevne applikasjoner

Strømmebehandling

Strømming av data over HTTP, også kjent som serversendte hendelser (SSE), er en mekanisme hvor serveren kontinuerlig sender data til klienten etter hvert som de blir tilgjengelige, uten at klienten eksplisitt må be om det. Siden KI-ens svar genereres trinnvis, er det fornuftig å gi en responsiv brukeropplevelse ved å vise KI-ens output etter hvert som det genereres. Og faktisk tilbyr alle KI-leverandørers API-er som jeg kjenner til, strømmende svar som et alternativ i deres fullføringsendepunkter.

Grunnen til at dette kapittelet kommer her i boken, rett etter Bruke verktøy, er på grunn av hvor kraftfullt det kan være å kombinere bruken av verktøy med direktesendte KI-svar til brukere. Dette muliggjør dynamiske og interaktive opplevelser hvor KI-en kan behandle brukerinndata, utnytte ulike verktøy og funksjoner etter eget skjønn, og gi sanntidssvar.

For å oppnå denne sømløse interaksjonen må du skrive strømhåndterere som kan distribuere både KI-påkalte verktøyfunksjonskall og klartekstutdata til sluttbrukeren.

Behovet for å gjenta løkken etter behandling av en verktøyfunksjon tilfører en interessant utfordring til oppgaven.

Implementering av en ReplyStream

For å demonstrere hvordan strømmebehandling kan implementeres, vil dette kapittelet ta en grundig gjennomgang av en forenklet versjon av `ReplyStream`-klassen som brukes i Olympia. Instanser av denne klassen kan sendes som `stream`-parameteren i KI-klientbiblioteker som ruby-openai og openrouter

Her er hvordan jeg bruker `ReplyStream` i Olympias `PromptSubscriber`, som lytter via Wisper etter opprettelsen av nye brukermeldinger.

```
1  class PromptSubscriber
2    include Raix::ChatCompletion
3    include Raix::PromptDeclarations
4
5    # many other declarations omitted...
6
7    prompt text: -> { user_message.content },
8           stream: -> { ReplyStream.new(self) },
9           until: -> { bot_message.complete? }
10
11   def message_created(message) # invoked by Wisper
12     return unless message.role.user? && message.content?
13
14     # rest of the implementation omitted...
```

I tillegg til en `context`-referanse til meldingsabonnenten som instansierte den, har klassen `ReplyStream` også instansvariabler for å lagre en buffer med mottatte data, og matriser for å holde oversikt over funksjonsnavn og argumenter som blir påkalt under strømbehandlingen.

```ruby
class ReplyStream
  attr_accessor :buffer, :f_name, :f_arguments, :context

  delegate :bot_message, :dispatch, to: :context

  def initialize(context)
    self.context = context
    self.buffer = []
    self.f_name = []
    self.f_arguments = []
  end

  def call(chunk, bytesize = nil)
    # ...
  end

  # ...
end
```

Metoden `initialize` setter opp starttilstanden for `ReplyStream`-instansen ved å initialisere bufferen, konteksten og andre variabler.

Metoden `call` er hovedinngangspunktet for behandling av strømmedataene. Den tar imot en `chunk` med data (representert som en hash) og en valgfri `bytesize`-parameter, som i vårt eksempel ikke brukes. Inne i denne metoden bruker klassen mønstergjenkjenning for å håndtere ulike scenarioer basert på strukturen til den mottatte chunken.

Ved å kalle `deep_symbolize_keys` på chunken gjør vi mønstergjenkjenningen mer elegant, ved at vi kan operere med symboler i stedet for strenger.

```
def call(chunk, _bytesize)
  case chunk.deep_symbolize_keys

  in { # match function name
    choices: [
      {
        delta: {
          tool_calls: [
            { index: index, function: {name: name} }
          ]
        }
      }
    ] }

    f_name[index] = name
```

Det første mønsteret vi ser etter er et verktøykall sammen med det tilhørende funksjonsnavnet. Hvis vi oppdager ett, legger vi det inn i `f_name`-arrayen. Vi lagrer funksjonsnavn i en indeksert array fordi modellen er i stand til å utføre parallelle funksjonskall, som sender mer enn én funksjon til utførelse samtidig.

> Parallell funksjonskalling er en AI-modells evne til å utføre flere funksjonskall sammen, noe som gjør det mulig å løse effektene og resultatene av disse funksjonskallene parallelt. Dette er spesielt nyttig hvis funksjoner tar lang tid, og reduserer antall kommunikasjonsrunder med API-et, som igjen kan spare betydelige mengder tokenforbruk.

Deretter må vi se etter argumentene som tilsvarer funksjonskallene.

```
  in { # match arguments
    choices: [
      {
        delta: {
          tool_calls: [
            {
              index: index, function: {arguments: argument }
            }
          ]
        }
      }
    ]}

    f_arguments[index] ||= "" # initialize if not already
    f_arguments[index] << argument
```

På samme måte som vi håndterte funksjonsnavnene, legger vi argumentene inn i en indeksert array.

Deretter ser vi etter brukerrettede meldinger, som vil komme fra serveren én token om gangen og bli tildelt `new_content`-variabelen. Vi må også holde øye med `finish_reason`. Den vil være `nil` frem til det siste segmentet av utdatasekvensen.

```
  in {
    choices: [
      { delta: {content: new_content}, finish_reason: finish_reason }
    ]}

    # you could transmit every chunk to the user here...
    buffer << new_content.to_s

    if finish_reason.present?
      finalize
    elsif new_content.to_s.match?(/\n\n/)
      send_to_client # ...or buffer and transmit once per paragraph
    end
```

Det er viktig at vi legger til et mønstergjenkjenningsuttrykk for å håndtere feilmeldinger sendt fra AI-modelleverandøren. I lokale utviklingsmiljøer kaster vi et unntak, men i produksjon logger vi feilen og avslutter.

```
  in { error: { message: } }
    if Rails.env.local?
      raise message
    else
      Honeybadger.notify("AI Error: #{message}")
      finalize
    end
```

Den siste else-delen av case vil kjøre hvis ingen av de foregående mønstrene ga treff. Det er bare en sikkerhetsmekanisme slik at vi oppdager det hvis AI-modellen begynner å sende oss ukjente deler.

```
  else
    Honeybadger.notify("Unrecognized Chunk: #{chunk}")
  end
end
```

Metoden `send_to_client` er ansvarlig for å sende det mellomlagrede innholdet til klienten. Den kontrollerer at bufferen ikke er tom, oppdaterer botmeldingsinnholdet, gjengir botmeldingen, og lagrer innholdet i databasen for å sikre datapersistens.

```
def send_to_client
  # no need to process pure whitespace
  return if buffer.join.squish.blank?

  # set the buffer content on the bot message
  content = buffer.join
  bot_message.content = content

  # save to database so that we never lose data
  # even if the stream doesn't terminate correctly
  bot_message.update_column(:content, content)

  # update content via websocket
  ConversationRenderer.update(bot_message)
end
```

`finalize`-metoden blir kalt når strømprosesseringen er fullført. Den utfører funksjonsanropene hvis noen ble mottatt under strømmen, oppdaterer botmeldingen

med det endelige innholdet og annen relevant informasjon, og nullstiller funksjonsanropshistorikken

```
def finalize
  if f_name.any?
    f_name.each_with_index do |name, index|
      # takes care of calling the function wherever it's implemented
      dispatch(name:, arguments: JSON.parse(f_arguments[index]))
    end

    # reset the function call history
    f_name.clear
    f_arguments.clear
  else
    content = buffer.join.presence
    bot_message.update!(content:, complete: true)
    ConversationRenderer.update(bot_message)
  end
end
```

Hvis modellen bestemmer seg for å kalle en funksjon, må du "ekspedere" dette funksjonskallet (navn og argumenter) på en slik måte at det blir utført og `function_call`- og `function_result`-meldinger blir lagt til i samtaleloggen

Basert på min erfaring er det bedre å håndtere opprettelsen av funksjonsmeldinger på ett sted i kodebasen din, i stedet for å stole på verktøyimplementasjonene. Det er ikke bare ryddigere, men har også en veldig viktig praktisk grunn: hvis AI-modellen kaller en funksjon, og ikke ser de resulterende kall- og resultatmeldingene i loggen når du går gjennom løkken, *vil den kalle den samme funksjonen igjen.* Potensielt for alltid. Husk at AI-en er fullstendig tilstandsløs, så med mindre du speiler disse funksjonskallene tilbake til den, har de ikke skjedd.

```ruby
# PromptSubscriber#dispatch

def dispatch(name:, arguments:)
  # adds a function_call message to the conversation transcript
  # plus dispatches to tool and returns result
  conversation.function_call!(name, arguments).then do |result|
    # add function result message to the transcript
    conversation.function_result!(name, result)
  end
end
```

Å tømme funksjonskallhistorikken etter utsending er like viktig som å sikre at kallet og resultatene havner i transkripsjonen din, slik at du ikke bare fortsetter å kalle de samme funksjonene om og om igjen hver gang du går gjennom løkken.

"Samtaleløkken"

Jeg nevner stadig løkker, men hvis du er ny til funksjonsanrop, er det kanskje ikke åpenbart *hvorfor* vi trenger å løkke. Grunnen er at når KI-en "ber" deg om å utføre verktøyfunksjoner på dens vegne, vil den stoppe å svare. Det er opp til deg å utføre disse funksjonene, samle resultatene, legge resultatene til i transkripsjonen, og deretter sende inn det opprinnelige promptet på nytt for å få et nytt sett med funksjonsanrop eller brukerrettede resultater.

I `PromptSubscriber`-klassen bruker vi `prompt`-metoden fra `PromptDeclarations`-modulen for å definere oppførselen til samtaleløkken. `until`-parameteren er satt til `-> { bot_message.complete? }`, som betyr at løkken vil fortsette til `bot_message` er markert som fullført.

```
1  prompt text: -> { user_message.content },
2         stream: -> { ReplyStream.new(self) },
3         until: -> { bot_message.complete? }
```

Men når blir `bot_message` markert som fullført? Hvis du har glemt det, se tilbake på linje 13 i `finalize`-metoden.

La oss gjennomgå hele strømmebehandlingslogikken.

1. `PromptSubscriber` mottar en ny brukermelding via `message_created`-metoden, som blir påkalt av Wisper pub/sub-systemet hver gang sluttbrukeren oppretter en ny prompt.
2. Klassemetoden `prompt` definerer deklarativt oppførselen til chatfullførelseslogikken for `PromptSubscriber`. AI-modellen vil utføre en chatfullførelse med brukerens meldingsinnhold, en ny instans av `ReplyStream` som strømparameter, og den spesifiserte løkkebetingelsen.
3. AI-modellen behandler prompten og begynner å generere et svar. Mens svaret strømmes, blir `call`-metoden til `ReplyStream`-instansen påkalt for hver datadel.
4. Hvis AI-modellen bestemmer seg for å kalle en verktøyfunksjon, blir funksjonsnavnet og argumentene hentet ut fra delen og lagret i henholdsvis `f_name`- og `f_arguments`-arrayene.
5. Hvis AI-modellen genererer brukerrettet innhold, blir det mellomlagret og sendt til klienten via `send_to_client`-metoden.
6. Når strømmebehandlingen er fullført, blir `finalize`-metoden kalt. Hvis noen verktøyfunksjoner ble påkalt under strømmen, blir de ekspedert ved hjelp av `dispatch`-metoden til `PromptSubscriber`.
7. `dispatch`-metoden legger til en `function_call`-melding i samtaleloggen, utfører den tilsvarende verktøyfunksjonen, og legger til en `function_result`-melding i loggen med resultatet av funksjonskallet.

8. Etter ekspederingen av verktøyfunksjonene blir funksjonskallhistorikken tømt for å forhindre dupliserte funksjonskall i påfølgende løkker.
9. Hvis ingen verktøyfunksjoner ble påkalt, oppdaterer `finalize`-metoden `bot_message` med det endelige innholdet, markerer det som fullført, og sender den oppdaterte meldingen til klienten.
10. Løkkebetingelsen `-> { bot_message.complete? }` blir evaluert. Hvis `bot_message` ikke er markert som fullført, fortsetter løkken, og den opprinnelige prompten sendes inn igjen med den oppdaterte samtaleloggen.
11. Trinn 3-10 gjentas til `bot_message` er markert som fullført, som indikerer at AI-modellen har fullført genereringen av svaret og ingen flere verktøyfunksjoner trenger å utføres.

Ved å implementere denne samtaleløkken, gjør du det mulig for AI-modellen å engasjere seg i en frem-og-tilbake-interaksjon med applikasjonen, utføre verktøyfunksjoner etter behov og generere brukerrettede svar til samtalen når en naturlig konklusjon.

Kombinasjonen av strømmebehandling og samtaleløkken muliggjør dynamiske og interaktive AI-drevne opplevelser, hvor AI-modellen kan behandle brukerinndata, utnytte ulike verktøy og funksjoner, og gi sanntidssvar basert på den utviklende samtalekonteksten.

Automatisk fortsettelse

Det er viktig å være klar over AI-utdatabegrensninger. De fleste modeller har et maksimalt antall tokens de kan generere i ett enkelt svar, som bestemmes av `max_tokens`-parameteren. Hvis AI-modellen når denne grensen mens den genererer et svar, vil den brått stoppe og indikere at utdataen ble avkortet.

I strømmesvaret fra AI-plattformens API kan du oppdage denne situasjonen ved å undersøke `finish_reason`-variabelen i delen. Hvis `finish_reason` er satt til

`"length"` (eller en annen nøkkelverdi spesifikk for modellen), betyr det at modellen nådde sin maksimale tokengrense under generering og utdataen har blitt kuttet kort.

En måte å håndtere dette scenariet på en elegant måte og gi en sømløs brukeropplevelse, er å implementere en automatisk fortsettelsesmekanisme i strømmebehandlingslogikken din. Ved å legge til et mønstertreff for lengderelaterte avslutningsgrunner, kan du velge å løkke og automatisk fortsette utdataen fra der den slapp.

Her er et med vilje forenklet eksempel på hvordan du kan modifisere `call`-metoden i `ReplyStream`-klassen for å støtte automatisk fortsettelse:

```
LENGTH_STOPS = %w[length MAX_TOKENS]

def call(chunk, _bytesize)
  case chunk.deep_symbolize_keys
    # ...

    in {
      choices: [
        { delta: {content: new_content},
          finish_reason: finish_reason } ] }

      buffer << new_content.to_s

      if finish_reason.blank?
        send_to_client if new_content.to_s.match?(/\n\n/)
      elsif LENGTH_STOPS.include?(finish_reason)
        continue_cutoff
      else
        finalize
      end

    # ...
  end
end

private

def continue_cutoff
  conversation.bot_message!(buffer.join, visible: false)
```

```
30    conversation.user_message!("please continue", visible: false)
31    bot_message.update_column(:created_at, Time.current)
32  end
```

I denne modifiserte versjonen, når `finish_reason` indikerer avkortet output, i stedet for å avslutte strømmen, legger vi til et par meldinger i transskriptet uten å avslutte, flytter den opprinnelige brukervendte responsmeldingen til "bunnen" av transskriptet ved å oppdatere dens `created_at`-attributt, og lar deretter løkken fortsette, slik at AI-en fortsetter å generere der den slapp.

Husk at AI-fullføringsendepunktet er tilstandsløst. Det "vet" bare det du forteller det via transskriptet. I dette tilfellet er måten vi kommuniserer til AI-en at den ble avkuttet på ved å legge til "usynlige" (for sluttbrukeren) meldinger i transskriptet. Husk imidlertid at dette er et bevisst forenklet eksempel. En faktisk implementering ville måtte utføre ytterligere transkripthåndtering for å sikre at vi ikke kastet bort tokens og/eller forvirret AI-en med dupliserte assistentmeldinger i transskriptet.

En faktisk implementering av autokontinuasjon bør også ha såkalt "kretsbryterlogikk" på plass for å forhindre ukontrollert løkking. Grunnen er at, gitt visse typer brukerprompts og lave `max_tokens`-innstillinger, kunne AI-en fortsette å løkke brukervendt output i det uendelige.

> Husk at hver løkke krever en separat forespørsel, og at hver forespørsel forbruker hele transskriptet ditt igjen. Du bør definitivt vurdere avveiningene mellom brukeropplevelse og API-bruk når du bestemmer deg for om du skal implementere autokontinuasjon i applikasjonen din. Autokontinuasjon kan være spesielt farlig dyrt, særlig når du bruker premium kommersielle modeller.

Konklusjon

Strømbehandling er et kritisk aspekt ved bygging av AI-drevne applikasjoner som kombinerer verktøybruk med direkte AI-responser. Ved å effektivt håndtere strømmedata fra AI-plattform-APIer, kan du gi en sømløs og interaktiv brukeropplevelse, håndtere store responser, optimalisere ressursbruk og håndtere feil på en elegant måte.

Den medfølgende `Conversation::ReplyStream`-klassen demonstrerer hvordan strømbehandling kan implementeres i en Ruby-applikasjon ved hjelp av mønstergjenkjenning og hendelsesdrevet arkitektur. Ved å forstå og utnytte strømbehandlingsteknikker kan du låse opp det fulle potensialet til AI-integrasjon i applikasjonene dine og levere kraftige og engasjerende brukeropplevelser.

Selvhelbredende data

Selvhelbredende data er en kraftfull tilnærming for å sikre dataintegritet, konsistens og kvalitet i applikasjoner ved å utnytte mulighetene som store språkmodeller (LLM-er) gir. Denne kategorien mønstre fokuserer på ideen om å bruke KI til automatisk å oppdage, diagnostisere og korrigere dataanomalier, inkonsistenser eller feil, og dermed redusere byrden på utviklere og opprettholde et høyt nivå av datapålitelighet.

I kjernen erkjenner mønstrene for selvhelbredende data at data er livsnerven i enhver applikasjon, og at det å sikre nøyaktighet og integritet er avgjørende for applikasjonens riktige funksjon og brukeropplevelse. Imidlertid kan håndtering og vedlikehold av datakvalitet være en kompleks og tidkrevende oppgave, spesielt når applikasjoner vokser i størrelse og kompleksitet. Det er her KI-ens kraft kommer inn i bildet.

I mønstrene for selvhelbredende data brukes KI-arbeidere til kontinuerlig å overvåke og analysere applikasjonens data. Disse modellene har evnen til å forstå og tolke mønstre,

relasjoner og anomalier i dataene. Ved å utnytte deres naturlige språkprosessering og forståelsesevner kan de identifisere potensielle problemer eller inkonsistenser i dataene og iverksette passende tiltak for å rette dem.

Prosessen med selvhelbredende data innebærer vanligvis flere viktige trinn:

1. **Dataovervåking**: KI-arbeidere overvåker kontinuerlig applikasjonens datastrømmer, databaser eller lagringssystemer, og ser etter tegn på anomalier, inkonsistenser eller feil. Alternativt kan du aktivere en KI-komponent som reaksjon på et unntak.
2. **Avviksdeteksjon**: Når et problem oppdages, analyserer KI-arbeideren dataene i detalj for å identifisere problemets spesifikke art og omfang. Dette kan innebære å oppdage manglende verdier, inkonsistente formater eller data som bryter med forhåndsdefinerte regler eller begrensninger.
3. **Diagnose og korrigering**: Når problemet er identifisert, bruker KI-arbeideren sin kunnskap og forståelse av datadomenet til å bestemme passende handlingsforløp. Dette kan innebære automatisk korrigering av data, utfylling av manglende verdier eller markering av problemet for menneskelig intervensjon hvis nødvendig.
4. **Kontinuerlig læring (valgfritt, avhengig av brukstilfelle)**: Etter hvert som KI-arbeideren møter og løser ulike dataproblemer, kan den produsere output som beskriver hva som skjedde og hvordan den responderte. Denne metadataen kan mates inn i læringsprosesser som gjør det mulig for deg (og kanskje den underliggende modellen, via finjustering) å bli mer effektiv over tid i å identifisere og løse dataanomalier.

Ved automatisk å oppdage og korrigere dataproblemer kan du sikre at applikasjonen din opererer med data av høy kvalitet og pålitelighet. Dette reduserer risikoen for at feil, inkonsistenser eller datarelaterte feil påvirker applikasjonens funksjonalitet eller brukeropplevelse.

Når du har KI-arbeidere som håndterer oppgaven med dataovervåking og korrigering, kan du fokusere innsatsen din på andre kritiske aspekter ved applikasjonen. Dette sparer tid og ressurser som ellers ville blitt brukt på manuell datarengjøring og vedlikehold. Faktisk blir manuell håndtering av datakvalitet stadig mer utfordrende etter hvert som applikasjonene vokser i størrelse og kompleksitet. Mønstrene for "Selvhelbredende data" skalerer effektivt ved å utnytte KI-ens kraft til å håndtere store datamengder og oppdage problemer i sanntid.

På grunn av sin natur kan KI-modeller tilpasse seg endrede datamønstre, skjemaer eller krav over tid med lite eller ingen tilsyn. Så lenge deres direktiver gir tilstrekkelig veiledning, spesielt angående tiltenkte resultater, kan applikasjonen din utvikle seg og håndtere nye datascenarier uten å kreve omfattende manuell intervensjon eller kodeendringer.

Mønstrene for selvhelbredende data samsvarer godt med de andre kategoriene av mønstre vi har diskutert, som "Mangfold av arbeidere". Selvhelbredende datafunksjonalitet kan sees på som en spesialisert type arbeider som fokuserer spesifikt på å sikre datakvalitet og integritet. Denne typen arbeider opererer sammen med andre KI-arbeidere, hvor hver bidrar til forskjellige aspekter av applikasjonens funksjonalitet.

Implementering av mønstre for selvhelbredende data i praksis krever nøye design og integrasjon av KI-modeller i applikasjonsarkitekturen. På grunn av risikoen for datatap og korrupsjon bør du definere klare retningslinjer for hvordan du vil bruke denne teknikken. Du bør også vurdere faktorer som ytelse, skalerbarhet og datasikkerhet.

Praktisk casestudie: Reparering av ødelagt JSON

En av de mest praktiske og praktiske måtene å utnytte selvhelbredende data på er også veldig enkel å forklare: reparering av ødelagt JSON.

Denne teknikken kan anvendes på den vanlige utfordringen med å håndtere ufullkomne eller inkonsistente data generert av LLM-er, som ødelagt JSON, og gir en tilnærming for automatisk oppdagelse og korrigering av disse problemene.

Hos Olympia møter jeg regelmessig scenarioer hvor LLMer genererer JSON-data som ikke er helt gyldig. Dette kan skje av ulike årsaker, som at LLMen legger til kommentarer før eller etter selve JSON-koden, eller introduserer syntaksfeil som manglende kommaer eller ikke-eskapterte doble anførselstegn. Disse problemene kan føre til parseringsfeil og forårsake forstyrrelser i applikasjonens funksjonalitet.

For å håndtere dette problemet har jeg implementert en praktisk løsning i form av en JsonFixer-klasse. Denne klassen følger "Selvhelbredende data"-mønsteret ved å ta den ødelagte JSONen som input og bruke en LLM til å reparere den, samtidig som den bevarer så mye informasjon og intensjon som mulig.

```
class JsonFixer
  include Raix::ChatCompletion

  def call(bad_json, error_message)
    raise "No data provided" if bad_json.blank? || error_message.blank?

    transcript << {
        system: "Consider user-provided JSON that generated a parse
                 exception. Do your best to fix it while preserving the
                 original content and intent as much as possible." }
    transcript << { user: bad_json }
    transcript << { assistant: "What is the error message?"}
    transcript << { user: error_message }
    transcript << { assistant: "Here is the corrected JSON\n```json\n" }

    self.stop = ["```"]

    chat_completion(json: true)
  end

  def model
    "mistralai/mixtral-8x7b-instruct:nitro"
  end
```

```
24 end
```

Legg merke til hvordan `JsonFixer` bruker Ventriloquist for å styre AI-ens responser.

Prosessen med selvhelbredende JSON-data fungerer som følger:

1. **JSON-generering**: En LLM brukes til å generere JSON-data basert på bestemte prompts eller krav. På grunn av LLM-enes natur vil den genererte JSON-en ikke alltid være perfekt gyldig. JSON-parseren vil selvfølgelig utløse en `ParserError` hvis du gir den ugyldig JSON.

```
1 begin
2   JSON.parse(llm_generated_json)
3 rescue JSON::ParserError => e
4   JsonFixer.new.call(llm_generated_json, e.message)
5 end
```

Merk at feilmeldingen også sendes til `JSONFixer`-kallet slik at den ikke trenger å gjøre fullstendige antakelser om hva som er galt med dataene, spesielt siden parseren ofte vil fortelle deg nøyaktig hva som er feil.

2. **LLM-basert Korreksjon**: `JSONFixer`-klassen sender den ødelagte JSON-en tilbake til en LLM, sammen med en spesifikk prompt eller instruksjon for å fikse JSON-en mens den bevarer den opprinnelige informasjonen og intensjonen så mye som mulig. LLM-en, som er trent på store mengder data og har forståelse av JSON-syntaks, forsøker å korrigere feilene og generere en gyldig JSON-streng. Responsbegrensning brukes for å begrense outputen fra LLM-en, og vi velger Mixtral 8x7B som AI-modellen, siden den er spesielt god for denne typen oppgave.

3. **Validering og Integrering**: Den reparerte JSON-strengen som returneres av LLM-en blir analysert av selve `JSONFixer`-klassen, fordi den kalte `chat_completion(json: true)`. Hvis den reparerte JSON-en består valideringen, blir den integrert tilbake i applikasjonens arbeidsflyt, slik at applikasjonen kan fortsette å behandle dataene sømløst. Den dårlige JSON-en har blitt "helbredet".

Selv om jeg har skrevet og omskrevet min egen `JSONFixer`-implementasjon flere ganger, tviler jeg på at den totale tiden investert i alle disse versjonene er mer enn en time eller to.

Merk at bevaring av intensjon er et nøkkelelement i ethvert selvhelbredende datamønster. Den LLM-baserte korrigeringsprosessen tar sikte på å bevare den opprinnelige informasjonen og intensjonen i den genererte JSON-en så mye som mulig. Dette sikrer at den reparerte JSON-en beholder sin semantiske betydning og kan brukes effektivt innenfor applikasjonens kontekst.

Denne praktiske implementeringen av "Selvhelbredende Data"-tilnærmingen i Olympia demonstrerer tydelig hvordan AI, spesielt LLM-er, kan utnyttes for å løse datautfordringer i den virkelige verden. Den viser styrken ved å kombinere tradisjonelle programmeringsteknikker med AI-kapabiliteter for å bygge robuste og effektive applikasjoner.

Postels Lov og "Selvhelbredende Data"-Mønsteret

"Selvhelbredende Data", som eksemplifisert av JSONFixer-klassen, samsvarer godt med prinsippet kjent som Postels Lov, også referert til som Robusthetsprinsippet. Postels Lov sier:

"Vær konservativ i det du gjør, vær liberal i det du aksepterer fra andre."

Dette prinsippet, opprinnelig formulert av Jon Postel, en pioner innen det tidlige

Internettet, understreker viktigheten av å bygge systemer som er tolerante overfor diverse eller til og med lett ukorrekte inndata, mens de opprettholder streng overholdelse av spesifiserte protokoller når de sender utdata.

I konteksten av "Selvhelbredende Data" legemliggjør JSONFixer-klassen Postels Lov ved å være liberal i å akseptere ødelagt eller ufullkommen JSON-data generert av LLM-er. Den avviser eller feiler ikke umiddelbart når den møter JSON som ikke strengt følger det forventede formatet. I stedet tar den en tolerant tilnærming og forsøker å fikse JSON-en ved hjelp av kraften i LLM-er.

Ved å være liberal i å akseptere ufullkommen JSON, demonstrerer JSONFixer-klassen robusthet og fleksibilitet. Den anerkjenner at data i den virkelige verden ofte kommer i ulike former og ikke alltid samsvarer med strenge spesifikasjoner. Ved å håndtere og korrigere disse avvikene på en elegant måte, sikrer klassen at applikasjonen kan fortsette å fungere problemfritt, selv i nærvær av ufullkomne data.

På den annen side følger JSONFixer-klassen også den konservative delen av Postels Lov når det gjelder output. Etter å ha fikset JSON-en ved hjelp av LLM-er, validerer klassen den korrigerte JSON-en for å sikre at den strengt samsvarer med det forventede formatet. Den opprettholder dataenes integritet og korrekthet før den sender dem videre til andre deler av applikasjonen. Denne konservative tilnærmingen garanterer at outputen fra JSONFixer-klassen er pålitelig og konsistent, og fremmer interoperabilitet og forhindrer spredning av feil.

Interessante fakta om Jon Postel:

- Jon Postel (1943-1998) var en amerikansk informatiker som spilte en avgjørende rolle i utviklingen av Internettet. Han var kjent som "Internettets Gud" for sine betydelige bidrag til de underliggende protokollene og standardene.
- Postel var redaktør for Request for Comments (RFC)-dokumentserien, som er en serie tekniske og organisatoriske notater om Internettet. Han forfattet eller medforfattet over 200 RFC-er, inkludert de grunnleggende protokollene som

TCP, IP og SMTP.

- I tillegg til hans tekniske bidrag, var Postel kjent for sin ydmyke og samarbeidsvillige tilnærming. Han trodde på viktigheten av å oppnå konsensus og jobbe sammen for å bygge et robust og interoperabelt nettverk.
- Postel tjenestegjorde som direktør for Computer Networks Division ved Information Sciences Institute (ISI) ved University of Southern California (USC) fra 1977 frem til sin altfor tidlige død i 1998.
- Som anerkjennelse for hans enorme bidrag, ble Postel posthumt tildelt den prestisjefylte Turing-prisen i 1998, ofte referert til som "Databehandlingens Nobelpris."

`JSONFixer`-klassen fremmer robusthet, fleksibilitet og interoperabilitet, som var kjerneverdier som Postel opprettholdt gjennom hele sin karriere. Ved å bygge systemer som er tolerante for ufullkommenheter mens de opprettholder streng overholdelse av protokoller, kan vi skape applikasjoner som er mer motstandsdyktige og tilpasningsdyktige i møte med virkelige utfordringer.

Hensyn og kontraindikasjoner

Anvendbarheten av selvhelbredende datatilnærminger er helt avhengig av hvilken type data applikasjonen din håndterer. Det er en grunn til at du kanskje ikke vil bare monkeypatch `JSON.parse` for automatisk å korrigere *alle JSON-parsing-feil* i applikasjonen din: ikke alle feil kan eller bør korrigeres automatisk.

Selvhelbredende er spesielt utfordrende når det er koblet til regulatoriske eller etterlevelses-krav relatert til datahåndtering og -behandling. Noen bransjer, som helsevesen og finans, har så strenge forskrifter angående dataintegritet og sporbarhet at enhver form for "black box" datakorrigering uten skikkelig tilsyn eller logging kan bryte disse forskriftene. Det er avgjørende å sikre at alle selvhelbredende datateknikker

du kommer opp med, er i samsvar med gjeldende juridiske og regulatoriske rammeverk.

Anvendelse av selvhelbredende datateknikker, spesielt de som involverer AI-modeller, kan også ha stor innvirkning på applikasjonsytelse og ressursbruk. Behandling av store datamengder gjennom AI-modeller for feildeteksjon og -korrigering kan være beregningsmessig intensivt. Det er viktig å vurdere avveiningene mellom fordelene med selvhelbredende data og de tilhørende ytelses- og ressurskostnadene.

La oss nå se nærmere på faktorene som er involvert i å bestemme når og hvor man skal anvende denne kraftige tilnærmingen.

Datakritikalitet

Når man vurderer anvendelsen av selvhelbredende datateknikker, er det avgjørende å vurdere kritikaliteten til dataene som behandles. Kritikalitetsnivået refererer til viktigheten og sensitiviteten til dataene i konteksten av applikasjonen din og dens forretningsdomene.

I noen tilfeller kan automatisk korrigering av datafeil være upassende, spesielt hvis dataene er svært sensitive eller har juridiske implikasjoner. Vurder for eksempel følgende scenarioer:

1. **Finansielle transaksjoner**: I finansapplikasjoner, som banksystemer eller handelsplattformer, er datanøyaktighet av største betydning. Selv mindre feil i finansielle data kan ha betydelige konsekvenser, som feil i kontobalanser, feildirigerte midler eller feilaktige handelsbeslutninger. I disse tilfellene kan automatiserte korreksjoner uten grundig verifisering og revisjon introdusere uakseptable risikoer.
2. **Medisinske journaler**: Helseapplikasjoner håndterer svært sensitive og konfidensielle pasientdata. Unøyaktigheter i medisinske journaler kan ha alvorlige implikasjoner for pasientsikkerhet og behandlingsbeslutninger. Automatisk modifisering av medisinske data uten skikkelig tilsyn og validering

av kvalifisert helsepersonell kan bryte med regulatoriske krav og sette pasientens velvære i fare.

3. **Juridiske dokumenter**: Applikasjoner som håndterer juridiske dokumenter, som kontrakter, avtaler eller rettsdokumenter, krever streng nøyaktighet og integritet. Selv mindre feil i juridiske data kan ha betydelige juridiske konsekvenser. Automatiserte korreksjoner i dette domenet er kanskje ikke passende, siden dataene ofte krever manuell gjennomgang og verifisering av juridiske eksperter for å sikre gyldighet og håndhevbarhet.

I disse kritiske datascenarioene overgår risikoene forbundet med automatiserte korreksjoner ofte de potensielle fordelene. Konsekvensene av å introdusere feil eller modifisere data feilaktig kan være alvorlige, og føre til økonomiske tap, juridisk ansvar eller til og med skade på personer.

Når man håndterer svært kritiske data, er det essensielt å prioritere manuelle verifiserings- og valideringsprosesser. Menneskelig tilsyn og ekspertise er avgjørende for å sikre nøyaktighet og integritet i dataene. Automatiserte selvhelbredende teknikker kan fortsatt brukes til å flagge potensielle feil eller uoverensstemmelser, men den endelige beslutningen om korreksjoner bør involvere menneskelig vurdering og godkjenning.

Det er imidlertid viktig å merke seg at ikke alle data i en applikasjon nødvendigvis har samme kritikalitetsnivå. Innenfor samme applikasjon kan det være delsett av data som er mindre sensitive eller har lavere konsekvenser hvis feil oppstår. I slike tilfeller kan selvhelbredende datateknikker anvendes selektivt på disse spesifikke datasettene, mens kritiske data forblir gjenstand for manuell verifisering.

Nøkkelen er å nøye vurdere kritikaliteten til hver datakategori i applikasjonen din og definere klare retningslinjer og prosesser for håndtering av korreksjoner basert på tilhørende risikoer og implikasjoner. Ved å skille mellom kritiske (dvs. hovedbøker, medisinske journaler) og ikke-kritiske data (dvs. postadresser, ressursadvarsler), kan du

finne en balanse mellom å utnytte fordelene med selvhelbredende datateknikker der det er hensiktsmessig og opprettholde streng kontroll og tilsyn der det er nødvendig.

Til syvende og sist bør beslutningen om å anvende selvhelbredende datateknikker på kritiske data tas i samråd med domeneeksperter, juridiske rådgivere og andre relevante interessenter. Det er essensielt å vurdere de spesifikke kravene, forskriftene og risikoene forbundet med applikasjonens data og tilpasse datakorrigeringsstrategiene deretter.

Feilalvorlighet

Når man anvender selvhelbredende datateknikker, er det viktig å vurdere alvorlighetsgraden og innvirkningen av datafeilene. Ikke alle feil er like alvorlige, og passende handlingsforløp kan variere avhengig av problemets alvorlighetsgrad.

Mindre uoverensstemmelser eller formateringsproblemer kan være egnet for automatisk korrigering. For eksempel kan en selvhelbredende dataarbeider som er satt til å fikse ødelagt JSON håndtere manglende kommaer eller ueskapterte doble anførselstegn uten å betydelig endre dataenes mening eller struktur. Disse typene feil er ofte enkle å korrigere og har minimal innvirkning på den generelle dataintegriteten.

Imidlertid kan mer alvorlige feil som fundamentalt endrer betydningen eller integriteten til dataene kreve en annen tilnærming. I slike tilfeller er kanskje ikke automatiserte korreksjoner tilstrekkelige, og menneskelig inngripen kan være nødvendig for å sikre nøyaktigheten og gyldigheten av dataene.

Det er her konseptet med å bruke selve AI-en til å hjelpe med å bestemme feilalvorlighet kommer inn i bildet. Ved å utnytte AI-modellenes muligheter kan vi designe selvhelbredende dataarbeidere som ikke bare korrigerer feil, men også vurderer alvorlighetsgraden av disse feilene og tar informerte beslutninger om hvordan de skal håndteres.

La oss for eksempel se på en selvhelbredende dataarbeider som er ansvarlig for å korrigere uoverensstemmelser i data som strømmer inn i en kundedatabase. Arbeideren

kan designes til å analysere dataene og identifisere potensielle feil, som manglende eller motstridende informasjon. I stedet for å automatisk korrigere alle feil, kan arbeideren utstyres med ytterligere verktøyskall som gjør det mulig å flagge alvorlige feil for menneskelig gjennomgang.

Her er et eksempel på hvordan dette kan implementeres:

```
class CustomerDataReviewer
  include Raix::ChatCompletion
  include Raix::FunctionDeclarations

  attr_accessor :customer

  function :flag_for_review, reason: { type: "string" } do |params|
    AdminNotifier.review_request(customer, params[:reason])
  end

  def initialize(customer)
    self.customer = customer
  end

  def call(customer_data)
    transcript << {
      system: "You are a customer data reviewer. Your task is to identify
        and correct inconsistencies in customer data.

        < additional instructions here... >

        If you encounter severe errors that require human review, use the
        `flag_for_review` tool to flag the data for manual intervention." }

    transcript << { user: customer.to_json }
    transcript << { assistant: "Reviewed/corrected data:\n```json\n" }

    self.stop = ["```"]

    chat_completion(json: true).then do |result|
      return if result.blank?

      customer.update(result)
    end
```

```
35    end
36  end
```

I dette eksemplet er `CustomerDataHealer`-arbeideren designet for å identifisere og korrigere uoverensstemmelser i kundedata. Igjen bruker vi Responsavgrensning og Ventriloquist for å få strukturert output. Viktig er det at arbeiderens systemdirektiv inkluderer instruksjoner om å bruke `flag_for_review`-funksjonen hvis alvorlige feil oppdages.

Når arbeideren behandler kundedataene, analyserer den dataene og forsøker å korrigere eventuelle uoverensstemmelser. Hvis arbeideren fastslår at feilene er alvorlige og krever menneskelig inngrep, kan den bruke `flag_for_review`-verktøyet for å flagge dataene og oppgi en grunn for flaggingen.

`chat_completion`-metoden kalles med `json: true` for å tolke de korrigerte kundedataene som JSON. Det er ingen mulighet for løkker etter et funksjonskalll, så resultatet vil være tomt hvis `flag_for_review` ble påkalt. Ellers blir kunden oppdatert med de gjennomgåtte og potensielt korrigerte dataene.

Ved å inkorporere vurdering av feilenes alvorlighetsgrad og muligheten til å flagge data for menneskelig gjennomgang, blir den selvhelbredende dataarbeideren mer intelligent og tilpasningsdyktig. Den kan håndtere mindre feil automatisk mens alvorlige feil eskaleres til menneskelige eksperter for manuell intervensjon.

De spesifikke kriteriene for å bestemme feilenes alvorlighetsgrad kan defineres i arbeiderens direktiv basert på domenekunnskap og forretningskrav. Faktorer som påvirkning på dataintegritet, potensialet for datatap eller -korrupsjon, og konsekvensene av feilaktige data kan vurderes når alvorlighetsgraden skal fastsettes.

Ved å utnytte AI til å vurdere feilenes alvorlighetsgrad og gi muligheter for menneskelig intervensjon, kan selvhelbredende datateknikker skape balanse mellom automatisering og opprettholdelse av datanøyaktighet. Denne tilnærmingen sikrer at mindre feil korrigeres effektivt mens alvorlige feil får nødvendig oppmerksomhet og ekspertise fra menneskelige kontrollører.

Domenekompleksitet

Når man vurderer anvendelsen av selvhelbredende datateknikker, er det viktig å evaluere kompleksiteten i datadomenene og reglene som styrer deres struktur og relasjoner. Domenets kompleksitet kan ha betydelig innvirkning på effektiviteten og gjennomførbarheten av automatiserte datakorreksjonstilnærminger.

Selvhelbredende datateknikker fungerer godt når dataene følger veldefinerte mønstre og begrensninger. I domener hvor datastrukturen er relativt enkel og relasjonene mellom dataelementer er ukompliserte, kan automatiske korreksjoner anvendes med høy grad av sikkerhet. For eksempel kan korrigering av formateringsproblemer eller håndhevelse av grunnleggende datatypebegrensninger ofte håndteres effektivt av selvhelbredende dataarbeidere.

Imidlertid øker utfordringene knyttet til automatisk datakorrigering når domenets kompleksitet øker. I domener med intrikat forretningslogikk, komplekse relasjoner mellom dataentiteter, eller domenespesifikke regler og unntak, kan selvhelbredende datateknikker ikke alltid fange opp nyansene og kan introdusere utilsiktede konsekvenser.

La oss se på et eksempel på et komplekst domene: et finansielt handelssystem. I dette domenet involverer dataene ulike finansielle instrumenter, markedsdata, handelsregler og regulatoriske krav. Relasjonene mellom ulike dataelementer kan være intrikate, og reglene som styrer datavaliditet og konsistens kan være svært spesifikke for domenet.

I et så komplekst domene ville en selvhelbredende dataarbeider som er satt til å korrigere uoverensstemmelser i handelsdata, måtte ha en dyp forståelse av de domenespesifikke reglene og begrensningene. Den måtte ta hensyn til faktorer som markedsreguleringer, handelsgrenser, risikoberegninger og oppgjørsprosedyrer. Automatiske korreksjoner i denne konteksten vil kanskje ikke alltid fange opp domenets fulle kompleksitet og kan utilsiktet introdusere feil eller bryte domenespesifikke regler.

For å håndtere utfordringene med domenekompleksitet kan selvhelbredende

datateknikker forbedres ved å inkorporere domenespesifikk kunnskap og regler i AI-modellene og arbeiderne. Dette kan oppnås gjennom teknikker som:

1. **Domenespesifikk Trening**: AI-modellene som brukes for selvhelbredende data kan dirigeres eller til og med finjusteres på domenespesifikke datasett som fanger opp det spesifikke domenets kompleksitet og regler. Ved å eksponere modellene for representative data og scenarioer, kan de lære mønstrene, begrensningene og unntakene som er spesifikke for domenet.
2. **Regelbaserte Begrensninger**: Selvhelbredende dataarbeidere kan utvides med eksplisitte regelbaserte begrensninger som koder domenespesifikk kunnskap. Disse reglene kan defineres av domeneeksperter og integreres i datakorrigeringsprosessen. AI-modellene kan da bruke disse reglene til å guide sine beslutninger og sikre overholdelse av domenespesifikke krav.
3. **Samarbeid med Domeneeksperter**: I komplekse domener er det avgjørende å involvere domeneeksperter i design og utvikling av selvhelbredende datateknikker. Domeneeksperter kan gi verdifull innsikt i dataenes kompleksitet, forretningsreglene og potensielle kanttilfeller. Deres kunnskap kan inkorporeres i AI-modellene og arbeiderne for å forbedre nøyaktigheten og påliteligheten til automatiske datakorreksjoner ved bruk av Menneske-i-løkken-mønstre.
4. **Inkrementell og Iterativ Tilnærming**: Når man håndterer komplekse domener, er det ofte fordelaktig å adoptere en inkrementell og iterativ tilnærming til selvhelbredende data. I stedet for å forsøke å automatisere korreksjoner for hele domenet på én gang, fokuser på spesifikke underdomener eller datakategorier hvor reglene og begrensningene er godt forstått. Utvid gradvis omfanget av selvhelbredende teknikker etter hvert som forståelsen av domenet vokser og teknikkene viser seg effektive.

Ved å ta hensyn til kompleksiteten i datadomenet og inkorporere domenespesifikk kunnskap i selvhelbredende datateknikker, kan du oppnå en balanse mellom automatisering og nøyaktighet. Det er viktig å erkjenne at selvhelbredende data

ikke er en universalløsning, og at tilnærmingen bør tilpasses de spesifikke kravene og utfordringene i hvert domene.

I komplekse domener kan en hybrid tilnærming som kombinerer selvhelbredende datateknikker med menneskelig ekspertise og tilsyn være mest effektiv. Automatiske korreksjoner kan håndtere rutinepregede og veldefinerte tilfeller, mens komplekse scenarioer eller unntak kan flagges for menneskelig gjennomgang og intervensjon. Denne samarbeidsbaserte tilnærmingen sikrer at fordelene med automatisering realiseres samtidig som man opprettholder nødvendig kontroll og nøyaktighet i komplekse datadomener.

Forklarbarhet og gjennomsiktighet

Forklarbarhet refererer til evnen til å forstå og tolke resonnementet bak beslutningene som tas av AI-modeller, mens gjennomsiktighet innebærer å gi klar innsikt i datakorreksjonsprosessen.

I mange sammenhenger må dataendringer være reviderbare og kunne rettferdiggjøres. Interessenter, inkludert forretningsbrukere, revisorer og regulerende organer, kan kreve forklaringer på hvorfor visse datakorreksioner ble gjort og hvordan AI-modellene kom fram til disse beslutningene. Dette er spesielt viktig i domener hvor datanøyaktighet og integritet har betydelige implikasjoner, som finans, helsevesen og juridiske saker.

For å imøtekomme behovet for forklarbarhet og gjennomsiktighet bør selvhelbredende datateknikker inkorporere mekanismer som gir innsikt i AI-modellenes beslutningsprosess. Dette kan oppnås gjennom ulike tilnærminger:

1. **Tankekjede**: Ved å be modellen forklare sin tenkning "høyt" før den gjør endringer i data, kan man lettere forstå beslutningsprosessen og generere menneskelig lesbare forklaringer for korreksjoner som er gjort. Kompromisset er litt mer kompleksitet i å skille forklaringen fra den strukturerte datautgangen, som kan håndteres ved…

2. **Generering av forklaringer**: Selvhelbredende dataarbeidere kan utstyres med evnen til å generere menneskelig lesbare forklaringer for korreksjoner de gjør. Dette kan oppnås ved å be modellen produsere sin beslutningsprosess som lett forståelige forklaringer *integrert i selve dataene*. For eksempel kan en selvhelbredende dataarbeider generere en rapport som fremhever de spesifikke datauoverensstemmelsene den identifiserte, korreksjonene den anvendte, og begrunnelsen bak disse korreksjonene.
3. **Funksjonsrelevans**: AI-modeller kan instrueres med informasjon om viktigheten av ulike funksjoner eller attributter i datakorreksjonsprosessen som del av deres direktiver. Disse direktivene kan i sin tur eksponeres for menneskelige interessenter. Ved å identifisere nøkkelfaktorene som påvirker modellens beslutninger, kan interessenter få innsikt i resonnementet bak korreksjonene og vurdere deres gyldighet.
4. **Logging og revisjon**: Implementering av omfattende loggførings- og revisjonsmekanismer er avgjørende for å opprettholde gjennomsiktighet i den selvhelbredende dataprosessen. Hver datakorreksjon som gjøres av AI-modeller bør logges, inkludert originaldata, korrigerte data og spesifikke handlinger som er utført. Dette revisjonssporet muliggjør retrospektiv analyse og gir en klar oversikt over endringene som er gjort i dataene.
5. **Menneske-i-løkken-tilnærming**: Inkorporering av en menneske-i-løkken-tilnærming kan forbedre forklarbarheten og gjennomsiktigheten i selvhelbredende datateknikker. Ved å involvere menneskelige eksperter i gjennomgang og validering av AI-genererte korreksjoner, kan organisasjoner sikre at korreksjonene er i tråd med domenekunnskap og forretningskrav. Menneskelig tilsyn legger til et ekstra lag med ansvarlighet og muliggjør identifisering av potensielle skjevheter eller feil i AI-modellene.
6. **Kontinuerlig overvåking og evaluering**: Regelmessig overvåking og evaluering av ytelsen til selvhelbredende datateknikker er essensielt for å opprettholde gjennomsiktighet og tillit. Ved å vurdere nøyaktigheten og effektiviteten til

AI-modellene over tid, kan organisasjoner identifisere eventuelle avvik eller anomalier og iverksette korrigerende tiltak. Kontinuerlig overvåking bidrar til å sikre at den selvhelbredende dataprosessen forblir pålitelig og på linje med ønskede resultater.

Forklarbarhet og gjennomsiktighet er kritiske hensyn ved implementering av selvhelbredende datateknikker. Ved å gi klare forklaringer for datakorreksioner, opprettholde omfattende revisjonsspor og involvere menneskelig tilsyn, kan organisasjoner bygge tillit til den selvhelbredende dataprosessen og sikre at endringene som gjøres i dataene er forsvarlige og på linje med forretningsmålene.

Det er viktig å finne en balanse mellom fordelene med automatisering og behovet for gjennomsiktighet. Mens selvhelbredende datateknikker kan betydelig forbedre datakvalitet og effektivitet, bør dette ikke gå på bekostning av å miste oversikt og kontroll over datakorreksjonsprosessen. Ved å designe selvhelbredende dataarbeidere med forklarbarhet og gjennomsiktighet i tankene, kan organisasjoner utnytte kraften i AI samtidig som de opprettholder det nødvendige nivået av ansvarlighet og tillit til dataene.

Utilsiktede konsekvenser

Mens selvhelbredende datateknikker har som mål å forbedre datakvalitet og konsistens, er det avgjørende å være oppmerksom på potensialet for utilsiktede konsekvenser. Automatiske korreksjoner kan, hvis de ikke er nøye utformet og overvåket, utilsiktet endre betydningen eller konteksten til dataene, noe som kan føre til nedstrøms problemer.

En av de primære risikoene ved selvhelbredende data er introduksjonen av skjevheter eller feil i datakorreksjonsprosessen. AI-modeller kan, som alle andre programvaresystemer, være utsatt for skjevheter som finnes i treningsdataene eller som introduseres gjennom utformingen av algoritmene. Hvis disse skjevhetene

ikke identifiseres og reduseres, kan de forplante seg gjennom den selvhelbredende dataprosessen og resultere i skjeve eller feilaktige datamodifikasjoner.

Ta for eksempel en selvhelbredende dataarbeider som har som oppgave å korrigere uoverensstemmelser i kunders demografiske data. Hvis AI-modellen har lært skjevheter fra historiske data, som å knytte bestemte yrker eller inntektsnivåer til spesifikke kjønn eller etnisiteter, kan den gjøre feilaktige antakelser og modifisere dataene på en måte som forsterker disse skjevhetene. Dette kan føre til unøyaktige kundeprofiler, feilrettede forretningsbeslutninger og potensielt diskriminerende utfall.

En annen potensiell utilsiktet konsekvens er tapet av verdifull informasjon eller kontekst under datakorreksjonsprosessen. Selvhelbredende datateknikker fokuserer ofte på å standardisere og normalisere data for å sikre konsistens. I noen tilfeller kan imidlertid de originale dataene inneholde nyanser, unntak eller kontekstuell informasjon som er viktig for å forstå hele bildet. Automatiserte korreksjoner som blindt håndhever standardisering kan utilsiktet fjerne eller tilsløre denne verdifulle informasjonen.

For eksempel, tenk deg en selvhelbredende dataarbeider som er ansvarlig for å korrigere uoverensstemmelser i medisinske journaler. Hvis arbeideren møter på en pasients sykehistorie med en sjelden tilstand eller en uvanlig behandlingsplan, kan den forsøke å normalisere dataene for å passe et mer vanlig mønster. Men ved å gjøre dette kan den miste de spesifikke detaljene og konteksten som er avgjørende for å representere pasientens unike situasjon nøyaktig. Dette tapet av informasjon kan ha alvorlige konsekvenser for pasientbehandling og medisinske beslutninger.

For å redusere risikoen for utilsiktede konsekvenser er det viktig å ta en proaktiv tilnærming når man designer og implementerer selvhelbredende datateknikker:

1. **Grundig testing og validering**: Før man implementerer selvhelbredende dataarbeidere i produksjon, er det avgjørende å grundig teste og validere deres oppførsel mot et mangfold av scenarioer. Dette inkluderer testing med representative datasett som dekker ulike kanttilfeller, unntak og potensielle

skjevheter. Grundig testing hjelper med å identifisere og håndtere eventuelle utilsiktede konsekvenser før de påvirker data i den virkelige verden.

2. **Kontinuerlig overvåking og evaluering**: Å implementere mekanismer for kontinuerlig overvåking og evaluering er essensielt for å oppdage og redusere utilsiktede konsekvenser over tid. Regelmessig gjennomgang av resultatene fra selvhelbredende dataprosesser, analyse av påvirkningen på nedstrømssystemer og beslutningstaking, og innhenting av tilbakemeldinger fra interessenter kan hjelpe med å identifisere eventuelle negative effekter og utløse tidlige korrigerende tiltak. Hvis organisasjonen din har operasjonelle dashbord, er det sannsynligvis en god idé å legge til lett synlige målinger relatert til automatiserte dataendringer. Å legge til alarmer koblet til store avvik fra normal dataendringsaktivitet er sannsynligvis en enda bedre idé!
3. **Menneskelig tilsyn og intervensjon**: Det er avgjørende å opprettholde menneskelig tilsyn og muligheten til å gripe inn i den selvhelbredende dataprosessen. Mens automatisering kan forbedre effektiviteten betydelig, er det viktig å ha menneskelige eksperter som gjennomgår og validerer korreksjoner gjort av AI-modeller, spesielt i kritiske eller sensitive domener. Menneskelig dømmekraft og domenekunnskap kan hjelpe med å identifisere og håndtere eventuelle utilsiktede konsekvenser som kan oppstå.

4. **Forklarbar AI (XAI) og åpenhet**: Som diskutert i forrige underkapittel, kan inkorporering av forklarbar AI-teknikker og sikring av åpenhet i den selvhelbredende dataprosessen hjelpe med å redusere utilsiktede konsekvenser. Ved å gi klare forklaringer for datakorreksjonene og opprettholde omfattende revisjonsspor, kan organisasjoner bedre forstå og spore resonnementet bak modifikasjonene gjort av AI-modeller.
5. **Inkrementell og iterativ tilnærming**: Å adoptere en inkrementell og iterativ tilnærming til selvhelbredende data kan hjelpe med å minimere risikoen for utilsiktede konsekvenser. I stedet for å anvende automatiserte korreksjoner på hele datasettet på én gang, start med en delmengde av data og utvid gradvis omfanget

etter hvert som teknikkene viser seg å være effektive og pålitelige. Dette muliggjør nøye overvåking og justering underveis, og reduserer påvirkningen av eventuelle utilsiktede konsekvenser.

6. **Samarbeid og tilbakemelding**: Å engasjere interessenter fra forskjellige domener og oppmuntre til samarbeid og tilbakemelding gjennom hele den selvhelbredende dataprosessen kan hjelpe med å identifisere og håndtere utilsiktede konsekvenser. Regelmessig innhenting av innspill fra domeneeksperter, databrukere og sluttbrukere kan gi verdifull innsikt i den virkelige påvirkningen av datakorreksjonene og fremheve eventuelle problemer som kan ha blitt oversett.

Ved å proaktivt håndtere risikoen for utilsiktede konsekvenser og implementere passende sikkerhetstiltak, kan organisasjoner utnytte fordelene med selvhelbredende datateknikker mens de minimerer potensielle negative effekter. Det er viktig å tilnærme seg selvhelbredende data som en iterativ og samarbeidende prosess, kontinuerlig overvåke, evaluere og forbedre teknikkene for å sikre at de er på linje med ønskede utfall og opprettholder dataenes integritet og pålitelighet.

Når man vurderer bruken av selvhelbredende datamønstre, er det essensielt å nøye evaluere disse faktorene og veie fordelene opp mot potensielle risikoer og begrensninger. I noen tilfeller kan en hybrid tilnærming som kombinerer automatiserte korreksjoner med menneskelig tilsyn og intervensjon være den mest hensiktsmessige løsningen.

Det er også verdt å merke seg at selvhelbredende datateknikker ikke bør ses på som en erstatning for robust datavalidering, inputsanitering og feilhåndteringsmekanismer. Disse grunnleggende praksisene forblir kritiske for å sikre dataintegritet og sikkerhet. Selvhelbredende data bør ses på som en komplementær tilnærming som kan forsterke og forbedre disse eksisterende tiltakene.

Til syvende og sist avhenger beslutningen om å bruke selvhelbredende datamønstre av de spesifikke kravene, begrensningene og prioriteringene i applikasjonen din. Ved

å nøye vurdere hensynene skissert ovenfor og tilpasse dem til applikasjonens mål og arkitektur, kan du ta informerte beslutninger om når og hvordan du effektivt kan utnytte selvhelbredende datateknikker.

Kontekstuell innholdsgenerering

Mønstre for kontekstuell innholdsgenerering utnytter kraften i store språkmodeller (LLM) for å generere dynamisk og kontekstspesifikt innhold i applikasjoner. Denne kategorien mønstre anerkjenner viktigheten av å levere personalisert og relevant innhold til brukere basert på deres spesifikke behov, preferanser og til og med tidligere og nåværende interaksjoner med applikasjonen.

> I denne tilnærmingen refererer "innhold" både til primærinnhold (dvs. blogginnlegg, artikler, osv.) og meta-innhold, som anbefalinger til primærinnhold.

Mønstre for kontekstuell innholdsgenerering kan spille en avgjørende rolle i å forbedre brukerengasjementnivåene dine, tilby skreddersydde opplevelser og automatisere innholdsskapende oppgaver både for deg og brukerne dine. Ved å bruke mønstrene vi

beskriver i dette kapittelet, kan du lage applikasjoner som genererer innhold dynamisk og tilpasser seg kontekst og input i sanntid.

Mønstrene fungerer ved å integrere LLM i applikasjonens output, fra brukergrensesnittet (noen ganger referert til som "chrome"), til e-poster og andre former for varsler, samt alle innholdgenereringsprosesser.

Når en bruker samhandler med applikasjonen eller utløser en spesifikk innholdsforespørsel, fanger applikasjonen opp relevant kontekst, som brukerpreferanser, tidligere interaksjoner eller spesifikke prompts. Denne kontekstuelle informasjonen mates deretter inn i LLM-en, sammen med eventuelle nødvendige maler eller retningslinjer, og brukes til å produsere tekstlig output som ellers måtte ha vært enten hardkodet, lagret i en database eller algoritmisk generert.

LLM-generert innhold kan ta ulike former, som personaliserte anbefalinger, dynamiske produktbeskrivelser, tilpassede e-postsvar eller til og med hele artikler eller blogginnlegg. En av de mest radikale bruksområdene for dette innholdet som jeg var pionér for for over et år siden, er dynamisk generering av UI-elementer som skjemaetiketter, verktøytips og andre typer forklarende tekst.

Personalisering

En av hovedfordelene med mønstre for kontekstuell innholdsgenerering er muligheten til å levere svært personaliserte opplevelser til brukere. Ved å generere innhold basert på brukerspesifikk kontekst, gjør disse mønstrene det mulig for applikasjoner å skreddersy innhold til individuelle brukeres interesser, preferanser og interaksjoner.

Personalisering handler om mer enn bare å sette inn en brukers navn i generisk innhold. Det innebærer å utnytte den rike konteksten som er tilgjengelig om hver bruker for å generere innhold som resonerer med deres spesifikke behov og ønsker. Denne konteksten kan omfatte en rekke faktorer, som:

1. **Brukerprofilinformasjon**: På det mest generelle nivået av denne teknikken kan demografiske data, interesser, preferanser og andre profilattributter brukes til å generere innhold som samsvarer med brukerens bakgrunn og karakteristikker.
2. **Atferdsdata**: En brukers tidligere interaksjoner med applikasjonen, som viste sider, klikket lenker eller kjøpte produkter, kan gi verdifull innsikt i deres atferd og interesser. Disse dataene kan brukes til å generere innholdsforslag som gjenspeiler deres engasjementsmønstre og forutser deres fremtidige behov.
3. **Kontekstuelle faktorer**: Brukerens nåværende kontekst, som deres plassering, enhet, tid på døgnet eller til og med været, kan påvirke innholdgenereringsprosessen. For eksempel kan en reiseapplikasjon ha en AI-arbeider som kan generere personaliserte anbefalinger basert på brukerens nåværende plassering og gjeldende værforhold.

Ved å utnytte disse kontekstuelle faktorene, gjør mønstre for kontekstuell innholdsgenerering det mulig for applikasjoner å levere innhold som føles skreddersydd for hver enkelt bruker. Dette nivået av personalisering har flere betydelige fordeler:

1. **Økt engasjement**: Personalisert innhold fanger brukernes oppmerksomhet og holder dem engasjert i applikasjonen. Når brukere føler at innholdet er relevant og snakker direkte til deres behov, er det mer sannsynlig at de bruker mer tid på å samhandle med applikasjonen og utforske dens funksjoner.
2. **Forbedret brukertilfredshet**: Personalisert innhold viser at applikasjonen forstår og bryr seg om brukerens unike behov. Ved å tilby innhold som er hjelpsomt, informativt og på linje med deres interesser, kan applikasjonen øke brukertilfredsheten og bygge en sterkere forbindelse med brukerne sine.
3. **Høyere konverteringsrater**: I sammenheng med e-handel eller markedsføringsapplikasjoner kan personalisert innhold ha betydelig innvirkning på konverteringsrater. Ved å presentere brukere med produkter, tilbud eller anbefalinger som er skreddersydd til deres preferanser og atferd, kan

applikasjonen øke sannsynligheten for at brukere utfører ønskede handlinger, som å gjøre et kjøp eller registrere seg for en tjeneste.

Produktivitet

Mønstre for kontekstuell innholdsgenerering kan betydelig øke visse typer produktivitet ved å redusere behovet for manuell innholdsgenerering og redigering i kreative prosesser. Ved å utnytte kraften i LLM-er, kan du generere høykvalitetsinnhold i stor skala, og spare tid og innsats som innholdsskaperne og utviklerne dine ellers måtte ha brukt på kjedelig manuelt arbeid.

Tradisjonelt har innholdsskapere måttet forske, skrive, redigere og formatere innhold for å sikre at det oppfyller applikasjonens krav og brukerens forventninger. Denne prosessen kan være tidkrevende og ressursintensiv, særlig når innholdsmengden vokser.

Med mønstre for kontekstuell innholdsgenerering kan imidlertid innholdsskapingsprosessen i stor grad automatiseres. LLM-er kan generere sammenhengende, grammatisk korrekt og kontekstuelt relevant innhold basert på gitte prompts og retningslinjer. Denne automatiseringen gir flere produktivitetsfordeler:

1. **Redusert manuelt arbeid**: Ved å delegere innholdsgenerering til LLM-er kan innholdsskapere fokusere på oppgaver på høyere nivå som innholdsstrategi, idéutvikling og kvalitetssikring. De kan gi nødvendig kontekst, maler og retningslinjer til LLM-en og la den håndtere selve innholdsgenereringen. Dette reduserer det manuelle arbeidet som kreves for skriving og redigering, noe som gjør at innholdsskapere kan være mer produktive og effektive.
2. **Raskere innholdsproduksjon**: LLM-er kan generere innhold mye raskere enn menneskelige skribenter. Med de riktige promptene og retningslinjene kan en LLM produsere flere innholdsdeler på få sekunder eller minutter. Denne hastigheten gjør det mulig for applikasjoner å generere innhold i et mye raskere tempo, og holde tritt med brukernes behov og det stadig skiftende digitale landskapet.

> Fører raskere innholdsproduksjon til en "allmenningens tragedie" der internett drukner i innhold som ingen leser? Dessverre mistenker jeg at svaret er ja.

3. **Konsistens og kvalitet**: LLM-er kan enkelt revidere innhold slik at det blir konsistent i stil, tone og kvalitet. Gitt klare retningslinjer og eksempler, kan visse typer applikasjoner (f.eks. nyhetsredaksjoner, PR osv.) sikre at deres menneskegenererte innhold samsvarer med deres merkevares stemme og møter de ønskede kvalitetsstandardene. Denne konsistensen reduserer behovet for omfattende redigering og revisjoner, noe som sparer tid og innsats i innholdsskapingsprosessen.
4. **Iterasjon og optimalisering**: Mønstre for kontekstuell innholdsgenerering muliggjør rask iterasjon og optimalisering av innhold. Ved å justere promptene, malene eller retningslinjene som gis til LLM-en, kan applikasjonene dine raskt generere varianter av innhold og teste ulike tilnærminger på en automatisert måte som aldri var mulig tidligere. Denne iterative prosessen tillater raskere eksperimentering og forbedring av innholdsstrategier, som fører til mer effektivt og engasjerende innhold over tid. Denne spesielle teknikken kan være en total game-changer for applikasjoner som e-handel som lever og dør basert på fluktfrekvens og engasjement

Det er viktig å merke seg at selv om mønstre for kontekstuell innholdsgenerering kan øke produktiviteten betydelig, eliminerer de ikke fullstendig behovet for menneskelig involvering. Innholdsskapere og redaktører spiller fortsatt en avgjørende rolle i å definere den overordnede innholdsstrategien, gi veiledning til LLM-en og sikre kvaliteten og egnetheten til det genererte innholdet.

Ved å automatisere de mer repetitive og tidkrevende aspektene ved innholdsproduksjon,

frigjør mønstre for kontekstuell innholdsgenerering verdifull menneskelig tid og ressurser som kan omdirigeres til oppgaver med høyere verdi. Denne økte produktiviteten gjør det mulig å levere mer personalisert og engasjerende innhold til brukere mens man optimaliserer arbeidsflyter for innholdsproduksjon.

Rask iterasjon og eksperimentering

Mønstre for kontekstuell innholdsgenerering gjør det mulig å raskt iterere og eksperimentere med forskjellige innholdsvariasjoner, noe som muliggjør raskere optimalisering og forbedring av innholdsstrategien din. Du kan generere flere versjoner av innhold på få sekunder, simpelthen ved å justere konteksten, malene eller retningslinjene som gis til modellen.

Denne raske iterasjonsevnen gir flere viktige fordeler:

1. **Testing og optimalisering**: Med muligheten til å generere innholdsvariasjoner raskt, kan du enkelt teste forskjellige tilnærminger og måle deres effektivitet. For eksempel kan du generere flere versjoner av en produktbeskrivelse eller markedsføringsmelding, hver tilpasset et spesifikt brukersegment eller kontekst. Ved å analysere brukerengasjementsmålinger, som klikkfrekvens eller konverteringsrate, kan du identifisere de mest effektive innholdsvariasjonene og optimalisere innholdsstrategien din deretter.

2. **A/B-testing**: Mønstre for kontekstuell innholdsgenerering muliggjør sømløs A/B-testing av innhold. Du kan generere to eller flere variasjoner av innhold og tilfeldig servere dem til forskjellige brukergrupper. Ved å sammenligne ytelsen til hver variasjon kan du avgjøre hvilket innhold som resonerer best med målgruppen din. Denne datadrevne tilnærmingen lar deg ta informerte beslutninger og kontinuerlig forbedre innholdet ditt for å maksimere brukerengasjement og oppnå ønskede resultater.

3. **Personaliseringseksperimenter**: Rask iterasjon og eksperimentering er spesielt verdifullt når det kommer til personalisering. Med mønstre for kontekstuell innholdsgenerering kan du raskt generere personaliserte innholdsvariasjoner basert på forskjellige brukersegmenter, preferanser eller atferd. Ved å eksperimentere med forskjellige personaliseringsstrategier kan du identifisere de mest effektive tilnærmingene for å engasjere individuelle brukere og levere skreddersydde opplevelser.
4. **Tilpasning til endrede trender**: Evnen til å iterere og eksperimentere raskt gjør det mulig å forbli smidig og tilpasse seg endrede trender og brukerpreferanser. Når nye emner, nøkkelord eller brukeratferd dukker opp, kan du raskt generere innhold som samsvarer med disse trendene. Ved å kontinuerlig eksperimentere og forbedre innholdet ditt, kan du holde deg relevant og opprettholde et konkurransefortrinn i det stadig utviklende digitale landskapet.
5. **Kostnadseffektiv eksperimentering**: Tradisjonell innholdseksperimentering innebærer ofte betydelig tid og ressurser, ettersom innholdsskapere må manuelt utvikle og teste ulike variasjoner. Med mønstre for kontekstuell innholdsgenerering er imidlertid kostnadene ved eksperimentering betydelig redusert. LLM-er kan generere innholdsvariasjoner raskt og i stor skala, slik at du kan utforske et bredt spekter av ideer og tilnærminger uten å pådra deg betydelige kostnader.

For å få mest mulig ut av rask iterasjon og eksperimentering, er det viktig å ha et veldefinert eksperimenteringsrammeverk på plass. Dette rammeverket bør inkludere:

- Klare mål og hypoteser for hvert eksperiment
- Passende målinger og sporingsmekanismer for å måle innholdets ytelse
- Segmenterings- og målrettingsstrategier for å sikre at relevante innholdsvariasjoner blir levert til de riktige brukerne
- Analyse- og rapporteringsverktøy for å utlede innsikt fra eksperimentelle data
- En prosess for å inkorporere læring og optimaliseringer i innholdsstrategien din

Ved å omfavne rask iterasjon og eksperimentering kan du kontinuerlig forbedre og optimalisere innholdet ditt, og sikre at det forblir engasjerende, relevant og effektivt i å oppnå applikasjonens mål. Denne smidige tilnærmingen til innholdsskapelse lar deg ligge i forkant og levere eksepsjonelle brukeropplevelser.

Skalerbarhet og effektivitet

Ettersom applikasjoner vokser og etterspørselen etter personalisert innhold øker, muliggjør kontekstuell innholdsgenerering effektiv skalering av innholdsproduksjon. LLM-er kan generere innhold for et stort antall brukere og kontekster samtidig, uten behov for en proporsjonal økning i menneskelige ressurser. Denne skalbarheten gjør det mulig for applikasjoner å levere personaliserte opplevelser til en voksende brukerbase uten å belaste innholdsskapingskapasiteten.

Merk at kontekstuell innholdsgenerering kan brukes effektivt til å internasjonalisere applikasjonen din "på sparket". Faktisk er det akkurat det jeg gjorde ved å bruke min Instant18n Gem for å levere Olympia på mer enn et halvt dusin språk, selv om vi er mindre enn ett år gamle.

AI-drevet lokalisering

Hvis du tillater meg å skryte et øyeblikk, tror jeg at mitt Instant18n-bibliotek for Rails-apper er et banebrytende eksempel på "Kontekstuell innholdsgenerering"-mønsteret i aksjon, som viser det transformative potensialet til AI i applikasjonsutvikling. Denne gem-en utnytter kraften i OpenAIs GPT store språkmodell for å revolusjonere måten internasjonalisering og lokalisering håndteres i Rails-applikasjoner.

Tradisjonelt innebærer internasjonalisering av en Rails-applikasjon manuell definering av oversettelsesnøkler og tilhørende oversettelser for hvert støttet språk. Denne

prosessen kan være tidkrevende, ressursintensiv og utsatt for inkonsistenser. Med Instant18n gem-en er imidlertid paradigmet for lokalisering fullstendig omdefinert.

Ved å integrere en stor språkmodell gjør Instant18n gem-en det mulig å generere oversettelser på sparket, basert på konteksten og betydningen av teksten. I stedet for å være avhengig av forhåndsdefinerte oversettelsesnøkler og statiske oversettelser, oversetter gem-en dynamisk tekst ved hjelp av AI-kraft. Denne tilnærmingen gir flere viktige fordeler:

1. **Sømløs lokalisering**: Med Instant18n gem-en trenger utviklere ikke lenger å manuelt definere og vedlikeholde oversettelsesfiler for hvert støttet språk. Gem-en genererer automatisk oversettelser basert på den gitte teksten og ønsket målspråk, noe som gjør lokaliseringsprosessen uanstrengt og sømløs.
2. **Kontekstuell nøyaktighet**: AI kan gis nok kontekst til å forstå nyansene i teksten som oversettes. Den kan ta hensyn til den omkringliggende konteksten, idiomer og kulturelle referanser for å generere oversettelser som er nøyaktige, naturlig lydende og kontekstuelt passende.
3. **Omfattende språkstøtte**: Instant18n gem-en utnytter den enorme kunnskapen og språklige kapasiteten til GPT, og muliggjør oversettelser til et omfattende utvalg av språk. Fra vanlige språk som spansk og fransk til mer obskure eller fiktive språk som klingon og alvisk, kan gem-en håndtere et bredt spekter av oversettelseskrav.
4. **Fleksibilitet og kreativitet**: Gem-en går utover tradisjonelle språkoversettelser og tillater kreative og ukonvensjonelle lokaliseringsalternativer. Utviklere kan oversette tekst til ulike stiler, dialekter eller til og med fiktive språk, noe som åpner for nye muligheter for unike brukeropplevelser og engasjerende innhold.
5. **Ytelsesoptimalisering**: Instant18n gem-en inkorporerer bufringmekanismer for å forbedre ytelsen og redusere belastningen ved gjentatte oversettelser. Oversatt tekst blir bufret, slik at påfølgende forespørsler om samme oversettelse kan betjenes raskt uten behov for redundante API-kall.

Instant18n gem-en eksemplifiserer kraften i "Kontekstuell innholdsgenerering"-mønsteret ved å utnytte AI til å generere lokalisert innhold dynamisk. Den viser hvordan AI kan integreres i kjernefunksjonaliteten til en Rails-applikasjon, og transformere måten utviklere tilnærmer seg internasjonalisering og lokalisering.

Ved å eliminere behovet for manuell oversettelseshåndtering og muliggjøre sanntidsoversettelser basert på kontekst, sparer Instant18n gem utviklere betydelig tid og innsats. Det lar dem fokusere på å bygge kjernefunksjonaliteten i applikasjonen sin, samtidig som lokaliseringsaspektet håndteres sømløst og nøyaktig.

Viktigheten av Brukertesting og Tilbakemelding

Til slutt, husk alltid viktigheten av brukertesting og tilbakemelding. Det er avgjørende å validere at kontekstuell innholdsgenerering møter brukerforventningene og samsvarer med applikasjonens mål. Fortsett å iterere og forbedre generert innhold basert på brukerinnsikt og analysedata. Hvis du genererer dynamisk innhold i stor skala som ville være umulig å validere manuelt av deg og teamet ditt, vurder å legge til tilbakemeldingsmekanismer som lar brukere rapportere innhold som er rart eller feil, sammen med en forklaring på hvorfor. Denne verdifulle tilbakemeldingen kan til og med mates til en KI-arbeider som har som oppgave å gjøre justeringer i komponenten som genererte innholdet!

Generative UI

Oppmerksomhet er så verdifullt i disse dager at effektivt brukerengasjement nå krever programvareopplevelser som ikke bare er sømløse og intuitive, men også høyt personaliserte for individuelle behov, preferanser og kontekster. Som et resultat står designere og utviklere stadig oftere overfor utfordringen med å skape brukergrensesnitt som kan tilpasse seg og imøtekomme hver brukers unike behov *i stor skala*.

Generative UI (GenUI) er en virkelig revolusjonerende tilnærming til design av brukergrensesnitt som utnytter kraften i store språkmodeller (LLMs) for å skape høyt personaliserte og dynamiske brukeropplevelser på sparket. Jeg ønsket å gi deg i det minste en innføring i GenUI i denne boken, fordi jeg mener at det er en av de grønneste mulighetene som for tiden eksisterer innen applikasjonsdesign og rammeverk. Jeg er overbevist om at dusinvis eller flere nye vellykkede kommersielle og åpen kildekode-prosjekter vil dukke opp i denne spesielle nisjen.

I kjernen kombinerer GenUI prinsippene for Kontekstuell Innholdsgenerering med avanserte AI-teknikker for å generere brukergrensesnittelementer, som tekst, bilder og layouter, dynamisk basert på en dyp forståelse av brukerens kontekst, preferanser og mål. GenUI gjør det mulig for designere og utviklere å skape grensesnitt som tilpasser seg og utvikler seg som respons på brukerinteraksjoner, og gir et nivå av personalisering som tidligere var uoppnåelig.

GenUI representerer en grunnleggende endring i måten vi tilnærmer oss design av brukergrensesnitt. I stedet for å designe for massene, lar GenUI oss designe for individet. Personalisert innhold og grensesnitt har potensial til å skape brukeropplevelser som resonerer med hver bruker på et dypere nivå, og øker engasjement, tilfredshet og lojalitet.

Som en banebrytende teknikk er overgangen til GenUI full av konseptuelle og praktiske utfordringer. Integrering av AI i designprosessen, sikring av at de genererte grensesnittene ikke bare er personaliserte, men også brukbare, tilgjengelige og på linje med den overordnede merkevaren og brukeropplevelsen - alt dette er utfordringer som gjør GenUI til en jakt for de få, ikke de mange. I tillegg reiser involveringen av AI spørsmål om datapersonvern, åpenhet og kanskje til og med etiske implikasjoner

Til tross for utfordringene har personaliserte opplevelser i stor skala kraft til å fullstendig transformere måten vi samhandler med digitale produkter og tjenester på. Det åpner muligheter for å skape inkluderende og tilgjengelige grensesnitt som imøtekommer brukernes mangfoldige behov, uavhengig av deres evner, bakgrunn eller preferanser.

I dette kapittelet skal vi utforske konseptet GenUI, undersøke noen definerende karakteristikker, viktige fordeler og potensielle utfordringer. Vi begynner med å vurdere den mest grunnleggende og tilgjengelige formen for GenUI: generering av tekstinnhold for ellers tradisjonelt designede og implementerte brukergrensesnitt.

Generering av tekst for brukergrensesnitt

Tekstelementer som eksisterer i applikasjonens grensesnittselementer, som skjemaetiketter, verktøytips og forklarende tekst, er vanligvis hardkodet inn i malene eller UI-komponentene, og gir en konsistent men generisk opplevelse for alle brukere. Ved å bruke mønstre for kontekstuell innholdsgenerering, kan du transformere disse statiske elementene til dynamiske, kontekstbevisste og personaliserte komponenter.

Personaliserte skjemaer

Skjemaer er en allestedsnærværende del av web- og mobilapplikasjoner, og fungerer som det primære middelet for å samle inn brukerinput. Tradisjonelle skjemaer presenterer imidlertid ofte en generisk og upersonlig opplevelse, med standard etiketter og felter som ikke alltid samsvarer med brukerens spesifikke kontekst eller behov. Brukere er mer tilbøyelige til å fullføre skjemaer som føles skreddersydd for deres behov og preferanser, noe som fører til høyere konverteringsrater og brukertilfredshet.

Det er imidlertid viktig å finne en balanse mellom personalisering og konsistens. Mens tilpasning av skjemaer til individuelle brukere kan være fordelaktig, er det avgjørende å opprettholde et nivå av gjenkjennelighet og forutsigbarhet. Brukere skal fortsatt kunne gjenkjenne og navigere i skjemaer enkelt, selv med personaliserte elementer.

Her er noen personaliserte skjemaideer til inspirasjon:

Kontekstuelle feltforslag

GenUI kan analysere brukerens tidligere interaksjoner, preferanser og data for å gi intelligente feltforslag som prediksjoner. For eksempel, hvis brukeren tidligere har lagt inn sin leveringsadresse, kan skjemaet automatisk fylle ut de relevante feltene med deres lagrede informasjon. Dette sparer ikke bare tid, men viser også at applikasjonen forstår og husker brukerens preferanser.

Vent litt, er ikke denne teknikken noe som kunne vært gjort uten å involvere AI? Selvfølgelig, men det fine med å drive denne typen funksjonalitet med AI er todelt: 1) hvor enkelt det kan være å implementere og 2) hvor robust det kan være etter hvert som brukergrensesnittet ditt endres og utvikles over tid.

La oss sette sammen en tjeneste for vårt teoretiske ordrehåndteringssystem, som forsøker å proaktivt fylle inn riktig leveringsadresse for brukeren.

```
class OrderShippingAddressSubscriber
  include Raix::ChatCompletion

  attr_accessor :order

  delegate :customer, to: :order

  DIRECTIVE = "You are a smart order processing assistant. Given the
  customer's order history, guess the most likely shipping address
  for the current order."

  def order_created(order)
    return unless order.pending? && order.shipping_address.blank?

    self.order = order

    transcript.clear
    transcript << { system: DIRECTIVE }
    transcript << { user: "Order History: #{order_history.to_json}" }
    transcript << { user: "Current Order: #{order.to_json}" }

    response = chat_completion
    apply_predicted_shipping_address(order, response)
  end

  private

  def apply_predicted_shipping_address(order, response)
    # extract the shipping address from the response...
    # ...and assume there's some sort of live update of the address fields
    order.update(shipping_address:)
  end
```

```
33
34    def order_history
35      customer.orders.successful.limit(100).map do |order|
36        {
37          date: order.date,
38          description: order.description,
39          shipping_address: order.shipping_address
40        }
41      end
42    end
43  end
```

Dette eksempelet er svært forenklet, men burde fungere i de fleste tilfeller. Ideen er å la AI-en gjette på samme måte som et menneske ville gjort. For å tydeliggjøre hva jeg snakker om, la oss se på noen eksempeldata:

```
1  Order History:
2  [
3    {"date": "2024-01-03", "description": "garden soil mix",
4     "shipping_address": "123 Country Lane, Rural Town"},
5    {"date": "2024-01-15", "description": "hardcover fiction novels",
6     "shipping_address": "456 City Apt, Metroville"},
7    {"date": "2024-01-22", "description": "baby diapers", "shipping_address":
8     "789 Suburb St, Quietville"},
9    {"date": "2024-02-01", "description": "organic vegetables",
10    "shipping_address": "123 Country Lane, Rural Town"},
11   {"date": "2024-02-17", "description": "mystery thriller book set",
12    "shipping_address": "456 City Apt, Metroville"},
13   {"date": "2024-02-25", "description": "baby wipes",
14    "shipping_address": "789 Suburb St, Quietville"},
15   {"date": "2024-03-05", "description": "flower seeds",
16    "shipping_address": "123 Country Lane, Rural Town"},
17   {"date": "2024-03-20", "description": "biographies",
18    "shipping_address": "456 City Apt, Metroville"},
19   {"date": "2024-03-30", "description": "baby formula",
20    "shipping_address": "789 Suburb St, Quietville"},
21   {"date": "2024-04-12", "description": "lawn fertilizer",
22    "shipping_address": "123 Country Lane, Rural Town"},
23   {"date": "2024-04-22", "description": "science fiction novels",
24    "shipping_address": "456 City Apt, Metroville"},
```

```
  {"date": "2024-05-02", "description": "infant toys",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-05-14", "description": "outdoor grill",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-05-29", "description": "literary classics",
  "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-06-11", "description": "baby clothes",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-07-01", "description": "watering can",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-07-18", "description": "non-fiction essays",
  "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-07-28", "description": "baby bath items",
  "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-08-09", "description": "herb garden kit",
  "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-08-24", "description": "children's books",
  "shipping_address": "456 City Apt, Metroville"}
]
```

La du merke til mønsteret i dataene? Jeg garanterer at dette er enkelt stoff for en LLM. For å demonstrere dette, la oss spørre GPT-4 hva som er den mest sannsynlige leveringsadressen for et "termometer".

```
From the order history you've provided, it looks like the purchases are
generally clustered into three main types based on the shipping addresses:

123 Country Lane, Rural Town - This address often orders garden and
outdoor-related items like soil mix, vegetables, flower seeds, lawn
fertilizer, an outdoor grill, a watering can, and an herb garden kit.

456 City Apt, Metroville - This address tends to order various types of
books, indicating an interest in reading across genres like fiction,
thrillers, biographies, science fiction, literary classics, non-fiction
essays, and children's books.

789 Suburb St, Quietville - The purchases for this address are consistently
baby-related products, such as diapers, wipes, baby formula, infant toys,
baby clothes, and baby bath items.

```

```
Given these patterns, a digital thermometer could potentially be a fit for
any of the addresses based on different needs: outdoor temperature tracking,
a tool for a passionate reader interested in science gadgets, or a health
tool for a baby. However, considering its common use in monitoring health,
especially for babies, it's most likely to be purchased by someone who's
already buying baby care products.

Therefore, the most likely shipping address for a digital thermometer, based
on the purchasing patterns, would be 789 Suburb St, Quietville.
```

Hvis du tenker at det er overkill å bruke en dyr modell som GPT-4 for denne oppgaven, har du rett! Jeg prøvde den samme prompten på Mistral 7B Instruct og den ga følgende svar med en hastighet på 75 token per sekund, til en beskjeden kostnad på $0.000218 USD.

```
Based on the given order history, it appears that the customer has received
orders for baby-related items (diapers, wipes, formula, clothes, bath items)
at the address "789 Suburb St, Quietville" and garden-related items (soil,
seeds, watering can, herb garden kit) at the address "123 Country Lane, Rural
Town". The customer has also received books (fiction, non-fiction, literary
classics, mystery-thriller, science fiction, biographies, children's books)
at the address "456 City Apt, Metroville".

Given that the new order is for a "thermometer", which doesn't seem to fit
into any of the categories that the customer has previously ordered, it's
hard to accurately predict the shipping address based solely on the order
history. However, given the 50%-50% split between baby-related and
garden-related items, it could somewhat lean towards the Baby-related items
address ("789 Suburb St, Quietville"). But remember, this is an assumption
and cannot be definitively confirmed without more context or information.
```

Er overheaden og kostnadene ved denne teknikken verdt det for å gjøre en betalingsopplevelse mer magisk? For mange nettbutikker, absolutt. Og ut fra hvordan det ser ut, kommer kostnadene for AI-behandling bare til å gå ned, spesielt for leverandører av åpen kildekode-modellverting i et kappløp mot bunnen.

Bruk en Prompt Template og StructuredIO sammen med Response Fencing for å optimalisere denne typen chatfullføring.

Adaptiv feltrekkefølge

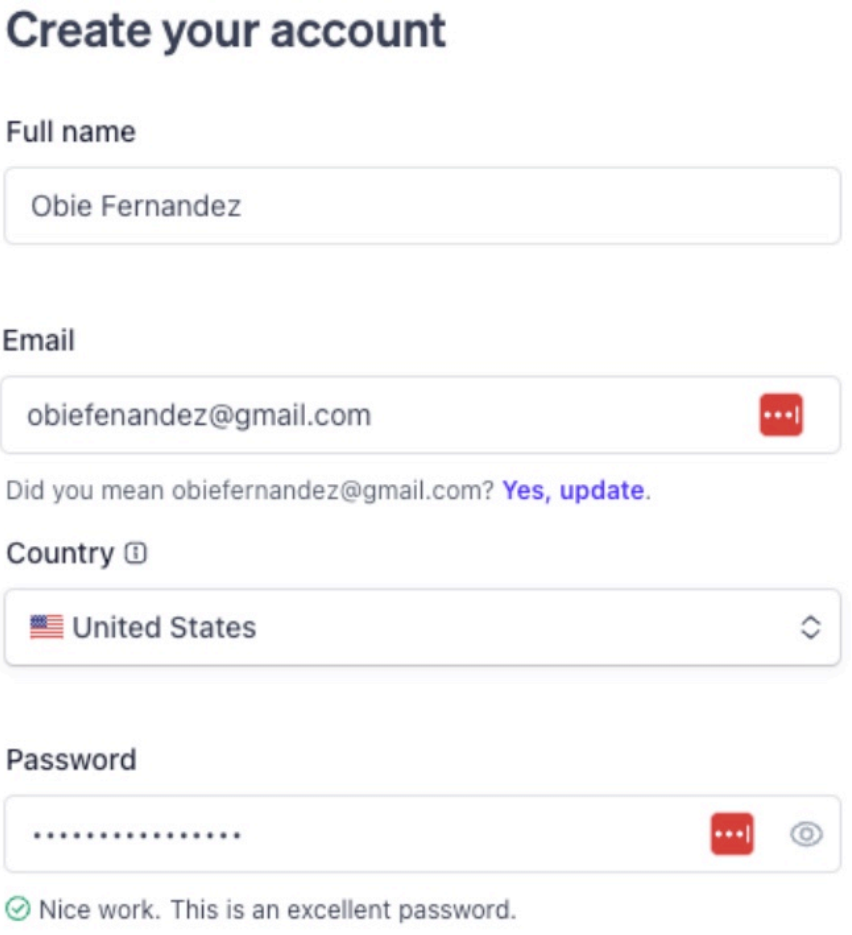
Rekkefølgen skjemafeltene presenteres i kan ha betydelig innvirkning på brukeropplevelsen og fullføringsgraden. Med GenUI kan du dynamisk justere feltrekkefølgen basert på brukerens kontekst og viktigheten av hvert felt. For eksempel, hvis brukeren fyller ut et registreringsskjema for en treningsapp, kan skjemaet prioritere felt relatert til deres treningsmål og preferanser, noe som gjør prosessen mer relevant og engasjerende.

Personalisert mikrotekst

Instruksjonsteksten, feilmeldinger og annen microcopy tilknyttet skjemaer kan også personaliseres ved hjelp av GenUI. I stedet for å vise generiske feilmeldinger som "Ugyldig e-postadresse," kan du generere mer hjelpsomme og kontekstuelle meldinger som "Vennligst skriv inn en gyldig e-postadresse for å motta ordrebekreftelsen din." Disse personlige tilpasningene kan gjøre skjemaopplevelsen mer brukervennlig og mindre frustrerende.

Personalisert validering

I samme gate som Personalisert mikrotekst, kunne du bruke AI til å validere skjemaet på måter som virker magiske. Forestill deg å la en AI validere et brukerprofilskjema, og se etter potensielle feil på et *semantisk* nivå.



Figur 9. Kan du se den semantiske valideringen som skjer?

Progressiv avsløring

GenUI kan intelligent bestemme hvilke skjemafelt som er essensielle basert på brukerens kontekst og gradvis avdekke flere felt etter behov. Denne progressive avsløringen bidrar til å redusere kognitiv belastning og gjør utfyllingsprosessen mer håndterbar. For eksempel, hvis en bruker registrerer seg for et grunnleggende abonnement, kan skjemaet

først presentere bare de essensielle feltene, og etterhvert som brukeren går videre eller velger spesifikke alternativer, kan ytterligere relevante felt introduseres dynamisk.

Kontekstbevisst forklarende tekst

Tooltips brukes ofte for å gi tilleggsinformasjon eller veiledning til brukere når de holder musepekeren over eller samhandler med spesifikke elementer. Med en "Kontekstuell innholdsgenerering"-tilnærming kan du generere tooltips som tilpasser seg brukerens kontekst og gir relevant informasjon. For eksempel, hvis en bruker utforsker en kompleks funksjon, kan tooltipet tilby personaliserte tips eller eksempler basert på deres tidligere interaksjoner eller ferdighetsnivå.

Forklarende tekst, som instruksjoner, beskrivelser eller hjelpemeldinger, kan genereres dynamisk basert på brukerens kontekst. I stedet for å presentere generiske forklaringer, kan du bruke LLM-er til å generere tekst som er skreddersydd til brukerens spesifikke behov eller spørsmål. For eksempel, hvis en bruker sliter med et bestemt trinn i en prosess, kan den forklarende teksten gi personalisert veiledning eller feilsøkingstips.

Microcopy refererer til de små tekstbitene som veileder brukere gjennom applikasjonen din, som knappeetiketter, feilmeldinger eller bekreftelsesmeldinger. Ved å anvende Kontekstuell innholdsgenerering-tilnærmingen på microcopy, kan du skape et adaptivt UI som reagerer på brukerens handlinger og gir relevant og hjelpsom tekst. For eksempel, hvis en bruker er i ferd med å utføre en kritisk handling, kan bekreftelsesmeldingen genereres dynamisk for å gi en klar og personalisert melding.

Personalisert forklarende tekst og tooltips kan i stor grad forbedre onboarding-prosessen for nye brukere. Ved å gi kontekstspesifikk veiledning og eksempler, kan du hjelpe brukere med å raskt forstå og navigere i applikasjonen, redusere læringskurven og øke adopsjonen.

Dynamiske og kontekstbevisste chrome-elementer kan også gjøre applikasjonen mer intuitiv og engasjerende. Brukere er mer tilbøyelige til å samhandle med og utforske

funksjoner når den medfølgende teksten er skreddersydd til deres spesifikke behov og interesser.

Så langt har vi dekket ideer for å forbedre eksisterende UI-paradigmer med KI, men hva med å tenke helt nytt om hvordan brukergrensesnitt designes og implementeres på en mer radikal måte?

Definering av Generativ UI

I motsetning til tradisjonell UI-design, hvor designere lager faste, statiske grensesnitt, peker GenUI mot en fremtid hvor programvaren vår har fleksible, personaliserte opplevelser som kan utvikle seg og tilpasse seg i sanntid. Hver gang vi bruker et KI-drevet samtalegrensesnitt, lar vi KI tilpasse seg brukerens spesifikke behov. GenUI tar dette et skritt videre ved å anvende dette nivået av tilpasningsevne på programvarens *visuelle* grensesnitt.

Grunnen til at det er mulig å eksperimentere med GenUI-ideer i dag, er at store språkmodeller allerede forstår programmering, og deres grunnleggende kunnskap inkluderer UI-teknologier og rammeverk. Spørsmålet er derfor om store språkmodeller kan brukes til å generere UI-elementer, som tekst, bilder, layouts og til og med hele grensesnitt, som er skreddersydd for hver enkelt bruker. Modellen kan instrueres til å ta hensyn til ulike faktorer, som brukerens tidligere interaksjoner, uttrykte preferanser, demografisk informasjon og den aktuelle brukskonteksten, for å skape svært personaliserte og relevante grensesnitt.

GenUI skiller seg fra tradisjonell brukergrensesnittdesign på flere viktige måter:

1. **Dynamisk og Adaptiv**: Tradisjonell UI-design innebærer å lage faste, statiske grensesnitt som forblir de samme for alle brukere. I motsetning til dette muliggjør

GenUI grensesnitt som kan dynamisk tilpasse og endre seg basert på brukerbehov og kontekst. Dette betyr at samme applikasjon kan presentere forskjellige grensesnitt til forskjellige brukere, eller til og med til samme bruker i forskjellige situasjoner.

2. **Personalisering i Stor Skala**: Med tradisjonell design er det ofte upraktisk å skape personaliserte opplevelser for hver bruker på grunn av tid og ressurser som kreves. GenUI, derimot, tillater personalisering i stor skala. Ved å utnytte KI kan designere skape grensesnitt som automatisk tilpasser seg hver brukers unike behov og preferanser, uten å måtte manuelt designe og utvikle separate grensesnitt for hvert brukersegment.
3. **Fokus på Resultater**: Tradisjonell UI-design fokuserer ofte på å skape visuelt tiltalende og funksjonelle grensesnitt. Mens disse aspektene fortsatt er viktige i GenUI, skifter hovedfokuset mot å oppnå ønskede brukerresultater. GenUI tar sikte på å skape grensesnitt som er optimalisert for hver brukers spesifikke mål og oppgaver, og prioriterer brukervennlighet og effektivitet fremfor rent estetiske hensyn.
4. **Kontinuerlig Læring og Forbedring**: GenUI-systemer kan kontinuerlig lære og forbedre seg over tid basert på brukerinteraksjoner og tilbakemeldinger. Når brukere interagerer med de genererte grensesnittene, kan KI-modellene samle data om brukeratferd, preferanser og resultater, og bruke denne informasjonen til å raffinere og optimalisere fremtidige grensesnittgenerasjoner. Denne iterative læringsprosessen gjør at GenUI-systemer blir stadig mer effektive i å møte brukerbehov over tid.

Det er viktig å merke seg at GenUI ikke er det samme som KI-assisterte designverktøy, som for eksempel de som gir forslag eller automatiserer visse designoppgaver. Mens disse verktøyene kan være nyttige for å effektivisere designprosessen, er de fortsatt avhengige av at designere tar endelige beslutninger og lager statiske grensesnitt. GenUI, derimot, innebærer at KI-systemet tar en mer aktiv rolle i selve genereringen og tilpasningen av grensesnitt basert på brukerdata og kontekst.

GenUI representerer et betydelig skifte i hvordan vi tilnærmer oss brukergrensesnittdesign, hvor vi beveger oss bort fra universalløsninger og mot høyt personaliserte, adaptive opplevelser. Ved å utnytte kraften i KI har GenUI potensial til å revolusjonere måten vi samhandler med digitale produkter og tjenester på, ved å skape grensesnitt som er mer intuitive, engasjerende og effektive for hver enkelt bruker.

Eksempel

For å illustrere konseptet med GenUI, la oss se på en hypotetisk treningsapplikasjon kalt "FitAI". Denne appen har som mål å gi personaliserte treningsplaner og kostholdsråd til brukere basert på deres individuelle mål, treningsnivå og preferanser.

I en tradisjonell UI-designtilnærming ville FitAI kanskje hatt et fast sett med skjermbilder og elementer som er like for alle brukere. Med GenUI kunne appens grensesnitt derimot dynamisk tilpasse seg hver brukers unike behov og kontekst.

Denne tilnærmingen er litt vanskelig å se for seg implementert i 2024 og har kanskje ikke engang tilstrekkelig ROI, men det er mulig.

Slik kunne det fungert:

1. **Onboarding**:
    - I stedet for et standard spørreskjema, bruker FitAI en konversasjons-KI for å samle informasjon om brukerens mål, nåværende treningsnivå og preferanser.
    - Basert på denne innledende interaksjonen genererer KI-en et personalisert dashbord-layout som fremhever funksjonene og informasjonen som er mest relevant for brukerens mål.
    - Dagens KI-teknologi kan ha et utvalg av skjermkomponenter til disposisjon for å komponere det personaliserte dashbordet.

- Fremtidig KI-teknologi kan ta på seg rollen som en erfaren UI-designer og faktisk skape dashbordet *fra bunnen av*.

2. **Treningsplanlegger**:

   - Treningsplanleggerens grensesnitt tilpasses av AI-en basert på brukerens erfaringsnivå og tilgjengelig utstyr.
   - For en nybegynner uten utstyr kan den vise enkle kroppsøvelser med detaljerte instruksjoner og videoer.
   - For en avansert bruker med tilgang til treningsstudio kan den vise mer komplekse treningsrutiner med mindre forklarende innhold.
   - Innholdet i treningsplanleggeren er ikke bare filtrert fra et stort datasett. Det kan genereres *fortløpende* basert på en kunnskapsbase som spørres med kontekst som inkluderer alt som er kjent om brukeren.

3. **Fremgangssporing**:

   - Fremgangssporingens grensesnitt utvikler seg basert på brukerens mål og engasjementsmønster.
   - Hvis en bruker primært fokuserer på vekttap, kan grensesnittet fremheve en vekttrendgraf og kaloriforbrenningsstatistikk.
   - For en bruker som bygger muskler, kan det fremheve styrkeøkning og endringer i kroppssammensetning.
   - AI-en kan tilpasse denne delen av applikasjonen til brukerens faktiske fremgang. Hvis fremgangen stopper opp i en periode, kan appen skifte til en modus hvor den prøver å få brukeren til å avsløre årsakene til tilbakeslaget, for å kunne motvirke dem.

4. **Ernæringsråd**:

   - Ernæringsdelen tilpasser seg brukerens kostholdspreferanser og -restriksjoner.

- For en vegansk bruker kan den vise plantebaserte måltidsforslag og proteinkilder.
- For en bruker med glutenintoleranse vil den automatisk filtrere ut glutenholdige matvarer fra anbefalingene.
- Igjen er innholdet ikke hentet fra et massivt datasett av måltidsdata som gjelder alle brukere, men er heller syntetisert fra en kunnskapsbase som inneholder informasjon som kan tilpasses basert på brukerens spesifikke situasjon og begrensninger.
- For eksempel genereres oppskrifter med ingrediensspesifikasjoner som matcher brukerens kontinuerlig endrede kaloribehov etter hvert som deres treningsnivå og kroppsstatistikk utvikler seg.

5. **Motivasjonselementer**:

   - Appens motivasjonsinnhold og varsler er personlig tilpasset basert på brukerens personlighetstype og respons på ulike motivasjonsstrategier.
   - Noen brukere kan motta oppmuntrende meldinger, mens andre får mer datadrevet tilbakemelding.

I dette eksempelet gjør GenUI det mulig for FitAI å skape en høyt tilpasset opplevelse for hver bruker, som potensielt øker engasjement, tilfredshet og sannsynligheten for å oppnå treningsmål. Grensesnittelementene, innholdet og til og med appens "personlighet" tilpasser seg for å best tjene hver enkelt brukers behov og preferanser.

Skiftet til Resultatorientert Design

GenUI representerer et fundamentalt skifte i tilnærmingen til brukergrensesnittdesign!, fra et fokus på å skape spesifikke grensesnittelementer til en mer helhetlig, resultatorientert tilnærming. Dette skiftet har flere viktige implikasjoner:

1. **Fokus på Brukermål**:

- Designere må tenke dypere på brukermål og ønskede resultater fremfor spesifikke grensesnittkomponenter.
- Vekten vil ligge på å skape systemer som kan generere grensesnitt som hjelper brukere å oppnå sine mål effektivt.
- Nye UI-rammeverk vil dukke opp som gir AI-baserte designere verktøyene de trenger for å kunne generere brukeropplevelser *fortløpende* og *fra bunnen av* istedenfor basert på forhåndsdefinerte skjermspesifikasjoner.

2. **Designernes Endrede Rolle**:

   - Designere vil gå over fra å lage faste layouter til å definere regler, begrensninger og retningslinjer som AI-systemer skal følge når de genererer grensesnitt.
   - De vil måtte utvikle ferdigheter innen områder som dataanalyse, AI prompt-teknikk og systemtenkning for å effektivt kunne veilede GenUI-systemer.

3. **Viktigheten av Brukerundersøkelser**:

   - Brukerundersøkelser blir enda mer kritisk i en GenUI-kontekst, ettersom designere må forstå ikke bare brukerpreferanser, men også hvordan disse preferansene og behovene endrer seg i ulike kontekster.
   - Kontinuerlig brukertesting og tilbakemeldingssløyfer vil være essensielt for å forbedre AI-ens evne til å generere effektive grensesnitt.

4. **Design for Variabilitet**:

   - Istedenfor å skape ett "perfekt" grensesnitt, må designere vurdere flere mulige variasjoner og sikre at systemet kan generere passende grensesnitt for ulike brukerbehov.
   - Dette inkluderer design for kanttilfeller og sikring av at de genererte grensesnittene opprettholder brukervennlighet og tilgjengelighet på tvers av ulike konfigurasjoner.

- Produktdifferensiering får nye dimensjoner som involverer divergerende perspektiver på brukerpsykologi og utnyttelse av unike datasett og kunnskapsbaser som ikke er tilgjengelige for konkurrenter.

Utfordringer og Hensyn

Mens GenUI tilbyr spennende muligheter, presenterer det også flere utfordringer og hensyn:

1. **Tekniske Begrensninger**:
    - Nåværende AI-teknologi, selv om den er avansert, har fortsatt begrensninger i å forstå komplekse brukerintensjoner og generere virkelig kontekstbevisste grensesnitt.
    - Ytelsesproblemer relatert til sanntidsgenerering av grensesnittelementer, spesielt på mindre kraftige enheter.
2. **Datakrav**:
    - Avhengig av bruksområdet kan effektive GenUI-systemer kreve betydelige mengder brukerdata for å generere personaliserte grensesnitt.
    - Utfordringene med etisk innhenting av autentiske brukerdata reiser bekymringer om datapersonvern og sikkerhet, samt potensielle skjevheter i dataene som brukes til å trene GenUI-modeller.
3. **Brukervennlighet og Konsistens**:
    - I hvert fall inntil praksisen blir utbredt, kan en applikasjon med konstant endrede grensesnitt føre til brukervennlighetsproblemer, ettersom brukere kan streve med å finne kjente elementer eller navigere effektivt.

- Det vil være avgjørende å finne balansen mellom personalisering og opprettholdelse av et konsistent, lærbart grensesnitt.

4. **Overavhengighet av AI**:

   - Det er en risiko for overdelegering av designbeslutninger til AI-systemer, som potensielt kan føre til uinspirerte, problematiske eller rett og slett ødelagte grensesnittvalg.
   - Menneskelig oversikt og muligheten til å overstyre AI-genererte design vil forbli viktig i overskuelig fremtid.

5. **Tilgjengelighetshensyn**:

   - Å sikre at dynamisk genererte grensesnitt forblir tilgjengelige for brukere med funksjonsnedsettelser presenterer helt nye utfordringer, noe som er bekymringsfullt gitt det dårlige nivået av tilgjengelighetssamsvar som typiske systemer viser.
   - På den annen side kan AI-designere implementeres med *innebygd* fokus på tilgjengelighet, og muligheter for å bygge tilgjengelige grensesnitt på sparket akkurat som de bygger UI for brukere uten funksjonsnedsettelser.
   - Uansett bør GenUI-systemer designes med robuste tilgjengelighetsretningslinjer og testprosesser.

6. **Brukertillit og Åpenhet**:

   - Brukere kan føle seg ukomfortable med grensesnitt som ser ut til å "vite for mye" om dem eller endrer seg på måter de ikke forstår.
   - Å gi åpenhet om hvordan og hvorfor grensesnitt personaliseres vil være viktig for å bygge brukertillit.

Fremtidsutsikter og Muligheter

Fremtiden for Generative UI (GenUI) har et enormt potensial for å revolusjonere måten vi samhandler med digitale produkter og tjenester på. Ettersom denne teknologien fortsetter å utvikle seg, kan vi forvente en omfattende endring i hvordan brukergrensesnitt designes, implementeres og oppleves. Jeg tror GenUI er fenomenet som endelig vil skyve programvaren vår inn i det som nå anses som science fiction.

En av de mest spennende mulighetene med GenUI er dets potensial til å forbedre tilgjengelighet i en skala som går utover å bare sikre at personer med alvorlige funksjonsnedsettelser ikke blir helt ekskludert fra bruken av programvaren din. Ved å automatisk tilpasse grensesnitt til individuelle brukerbehov, kan GenUI gjøre digitale opplevelser mer inkluderende enn noensinne. Tenk deg grensesnitt som sømløst justerer seg for å gi større tekst for yngre eller synshemmede brukere, eller forenklede layouter for de med kognitive funksjonsnedsettelser, alt uten å kreve manuell konfigurasjon eller separate "tilgjengelige" versjoner av applikasjoner.

Personaliseringsmulighetene til GenUI vil sannsynligvis drive økt brukerengasjement, tilfredshet og lojalitet på tvers av et bredt spekter av digitale produkter. Ettersom grensesnitt blir mer tilpasset individuelle preferanser og atferd, vil brukere finne digitale opplevelser mer intuitive og behagelige, noe som potensielt kan føre til dypere og mer meningsfylte interaksjoner med teknologi.

GenUI har også potensial til å transformere onboarding-prosessen for nye brukere. Ved å skape intuitive, personaliserte førstegangsopplevelser som raskt tilpasser seg hver brukers ekspertisenivå, kan GenUI betydelig redusere læringskurven forbundet med nye applikasjoner. Dette kan føre til raskere adopsjonsrater og økt brukerselvsikkerhet i utforskningen av nye funksjoner og funksjonalitet.

En annen spennende mulighet er GenUIs evne til å opprettholde en konsistent brukeropplevelse på tvers av forskjellige enheter og plattformer, samtidig som det optimaliseres for hver spesifikk brukskontekst. Dette kan løse den langvarige

utfordringen med å gi sammenhengende opplevelser på tvers av et stadig mer fragmentert enhetslandskap, fra smarttelefoner og nettbrett til stasjonære datamaskiner og fremvoksende teknologier som briller for utvidet virkelighet.

Den datadrevne naturen til GenUI åpner muligheter for rask iterasjon og forbedring i UI-design. Ved å samle sanntidsdata om hvordan brukere samhandler med genererte grensesnitt, kan designere og utviklere få enestående innsikt i brukeratferd og preferanser. Denne tilbakemeldingssløyfen kan føre til kontinuerlige forbedringer i UI-design, drevet av faktiske bruksmønstre heller enn antakelser eller begrenset brukertesting.

For å forberede seg på denne endringen må designere utvikle sine ferdigheter og tankesett. Fokuset vil skifte fra å lage faste layouter til å utvikle omfattende designsystemer og retningslinjer som kan informere AI-drevet grensesnittgenerering. Designere vil trenge å utvikle en dyp forståelse av dataanalyse, AI-teknologier og systemtenkning for å effektivt veilede GenUI-systemer.

Dessuten, ettersom GenUI visker ut grensene mellom design og teknologi, vil designere måtte samarbeide tettere med utviklere og dataforskere. Denne tverrfaglige tilnærmingen vil være avgjørende for å skape GenUI-systemer som ikke bare er visuelt tiltalende og brukervennlige, men også teknisk robuste og etisk forsvarlige.

De etiske implikasjonene av GenUI vil også komme i forgrunnen etter hvert som teknologien modnes. Designere vil spille en avgjørende rolle i utviklingen av rammeverk for ansvarlig bruk av kunstig intelligens i grensesnittdesign, for å sikre at personalisering forbedrer brukeropplevelsen uten å kompromittere personvern eller manipulere brukeratferd på uetiske måter.

Når vi ser mot fremtiden, representerer GenUI både spennende muligheter og betydelige utfordringer. Det har potensial til å skape mer intuitive, effektive og tilfredsstillende digitale opplevelser for brukere over hele verden. Selv om det vil kreve at designere tilpasser seg og tilegner seg nye ferdigheter, gir det også en enestående mulighet til å forme fremtiden for menneske-maskin-interaksjon på dyptgripende og meningsfylte

måter. Reisen mot fullt utviklede GenUI-systemer vil utvilsomt være kompleks, men de potensielle fordelene i form av forbedrede brukeropplevelser og digital tilgjengelighet gjør det til en fremtid det er verdt å strebe etter.

Intelligent arbeidsflytorkestrering

Innen applikasjonsutvikling spiller arbeidsflyter en avgjørende rolle i å definere hvordan oppgaver, prosesser og brukerinteraksjoner struktureres og utføres. Ettersom applikasjoner blir mer komplekse og brukerforventningene fortsetter å øke, blir behovet for intelligent og tilpasningsdyktig arbeidsflytorkestrering stadig mer åpenbart.

Tilnærmingen "Intelligent arbeidsflytorkestrering" fokuserer på å utnytte AI-komponenter for dynamisk å orkestrere og optimalisere komplekse arbeidsflyter i applikasjoner. Målet er å skape applikasjoner som er mer effektive, responsive og tilpasningsdyktige til sanntidsdata og kontekst.

I dette kapittelet skal vi utforske nøkkelprinsippene og mønstrene som understøtter tilnærmingen til intelligent arbeidsflytorkestrering. Vi vil vurdere hvordan AI kan

brukes til intelligent ruting av oppgaver, automatisert beslutningstaking og dynamisk tilpasning av arbeidsflyter basert på ulike faktorer som brukeratferd, systemytelse og forretningsregler. Gjennom praktiske eksempler og virkelige scenarier vil vi demonstrere det transformative potensialet AI har for å effektivisere og optimalisere applikasjonsarbeidsflyter.

Enten du bygger bedriftsapplikasjoner med komplekse forretningsprosesser eller forbrukerrettede applikasjoner med dynamiske brukerreiser, vil mønstrene og teknikkene som diskuteres i dette kapittelet gi deg kunnskapen og verktøyene til å skape intelligente og effektive arbeidsflyter som forbedrer den generelle brukeropplevelsen og skaper forretningsverdi.

Forretningsmessig behov

Tradisjonelle tilnærminger til arbeidsflythåndtering er ofte avhengige av forhåndsdefinerte regler og statiske beslutningstrær, som kan være rigide, infleksible og ute av stand til å håndtere den dynamiske naturen til moderne applikasjoner.

Tenk på et scenario hvor en e-handelsapplikasjon må håndtere en kompleks ordreoppfyllelsesprosess. Arbeidsflyten kan involvere flere trinn som ordrevalidering, lagerkontroll, betalingsbehandling, forsendelse og kundevarsling. Hvert trinn kan ha sitt eget sett med regler, avhengigheter, eksterne integrasjoner og unntakshåndteringsmekanismer. Å administrere en slik arbeidsflyt manuelt eller gjennom hardkodet logikk kan raskt bli tungvint, feilutsatt og vanskelig å vedlikeholde.

Dessuten, ettersom applikasjonen skalerer og antallet samtidige brukere vokser, kan arbeidsflyten måtte tilpasse og optimalisere seg selv basert på sanntidsdata og systemytelse. For eksempel kan applikasjonen under perioder med høy trafikk måtte dynamisk justere arbeidsflyten for å prioritere visse oppgaver, allokere ressurser effektivt og sikre en smidig brukeropplevelse.

Det er her tilnærmingen “Intelligent arbeidsflytorkestrering” kommer inn i bildet.

Ved å utnytte AI-komponenter kan utviklere skape arbeidsflyter som er intelligente, tilpasningsdyktige og selvoptimaliserende. AI kan analysere store mengder data, lære av tidligere erfaringer og ta informerte beslutninger i sanntid for å orkestrere arbeidsflyten effektivt.

Sentrale fordeler

1. **Økt effektivitet**: AI kan optimalisere oppgaveallokering, ressursutnyttelse og arbeidsflytutførelse, noe som fører til raskere prosesseringstider og forbedret total effektivitet.
2. **Tilpasningsevne**: AI-drevne arbeidsflyter kan dynamisk tilpasse seg endrede forhold, som svingninger i brukeretterspørsel, systemytelse eller forretningskrav, og sikre at applikasjonen forblir responsiv og robust.
3. **Automatisert beslutningstaking**: AI kan automatisere komplekse beslutningsprosesser innen arbeidsflyten, redusere manuell intervensjon og minimere risikoen for menneskelige feil.
4. **Personalisering**: AI kan analysere brukeratferd, preferanser og kontekst for å personalisere arbeidsflyten og levere skreddersydde opplevelser til individuelle brukere.
5. **Skalerbarhet**: AI-drevne arbeidsflyter kan skalere sømløst for å håndtere økende volumer av data og brukerinteraksjoner, uten å kompromittere ytelse eller pålitelighet.

I de følgende seksjonene vil vi utforske nøkkelmønstrene og teknikkene som muliggjør implementering av intelligente arbeidsflyter og vise frem virkelige eksempler på hvordan AI transformerer arbeidsflythåndtering i moderne applikasjoner.

Nøkkelmønstre

For å implementere intelligent arbeidsflytorkestrering i applikasjoner kan utviklere utnytte flere nøkkelmønstre som utnytter kraften i AI. Disse mønstrene gir en strukturert tilnærming til design og håndtering av arbeidsflyter, som gjør det mulig for applikasjoner å tilpasse seg, optimalisere og automatisere prosesser basert på sanntidsdata og kontekst. La oss utforske noen av de grunnleggende mønstrene i intelligent arbeidsflytorkestrering.

Dynamisk oppgaveruting

Dette mønsteret innebærer bruk av AI for intelligent ruting av oppgaver innen en arbeidsflyt basert på ulike faktorer som oppgaveprioritet, ressurstilgjengelighet og systemytelse. AI-algoritmer kan analysere karakteristikkene til hver oppgave, vurdere systemets nåværende tilstand og ta informerte beslutninger for å tildele oppgaver til de mest egnede ressursene eller prosesseringsveiene. Dynamisk oppgaveruting sikrer at oppgaver distribueres og utføres effektivt, og optimaliserer den totale arbeidsflytytelsen.

```
class TaskRouter
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  attr_accessor :task

  # list of functions that can be called by the AI entirely at its
  # discretion depending on the task received

  function :analyze_task_priority do
    TaskPriorityAnalyzer.perform(task)
  end

  function :check_resource_availability, # ...
  function :assess_system_performance, # ...
  function :assign_task_to_resource, # ...

```

```
18    DIRECTIVE = "You are a task router, responsible for intelligently
19     assigning tasks to available resources based on priority, resource
20     availability, and system performance..."
21
22    def initialize(task)
23      self.task = task
24      transcript << { system: DIRECTIVE }
25      transcript << { user: task.to_json }
26    end
27
28    def perform
29      while task.unassigned?
30        chat_completion
31
32        # todo: add max loop counter and break
33      end
34
35      # capture the transcript for later analysis
36      task.update(routing_transcript: transcript)
37    end
38  end
```

Merk løkken som er laget av `while`-uttrykket på linje 29, som fortsetter å spørre AI-en til oppgaven er tildelt. På linje 35 lagrer vi transkripsjonen av oppgaven for senere analyse og feilsøking, hvis det skulle bli nødvendig.

Kontekstuell beslutningstaking

Du kan bruke svært lignende kode for å ta kontekstbevisste beslutninger i en arbeidsflyt. Ved å analysere relevante datapunkter som brukerpreferanser, historiske mønstre og sanntidsinndata, kan AI-komponenter bestemme den mest hensiktsmessige handlingsplanen ved hvert beslutningspunkt i arbeidsflyten. Tilpass arbeidsflytens oppførsel basert på den spesifikke konteksten for hver bruker eller scenario, og gi personaliserte og optimaliserte opplevelser.

Adaptiv arbeidsflytkomposisjon

Dette mønsteret fokuserer på dynamisk sammensetting og justering av arbeidsflyter basert på endrede krav eller forhold. AI kan analysere arbeidsflytens nåværende tilstand, identifisere flaskehalser eller ineffektivitet, og automatisk modifisere arbeidsflytstrukturen for å optimalisere ytelsen. Adaptiv arbeidsflytkomposisjon gjør det mulig for applikasjoner å kontinuerlig utvikle og forbedre sine prosesser uten å kreve manuell intervensjon.

Unntakshåndtering og gjenoppretting

Unntakshåndtering og gjenoppretting er kritiske aspekter ved intelligent arbeidsflytorkestrering. Når man jobber med AI-komponenter og komplekse arbeidsflyter, er det essensielt å forutse og håndtere unntak på en elegant måte for å sikre systemets stabilitet og pålitelighet.

Her er noen viktige hensyn og teknikker for unntakshåndtering og gjenoppretting i intelligente arbeidsflyter:

1. **Unntakspropagering**: Implementer en konsistent tilnærming for å propagere unntak på tvers av arbeidsflytkomponenter. Når et unntak oppstår innenfor en komponent, bør det fanges opp, logges og propageres til orkestratoren eller en separat komponent som er ansvarlig for å håndtere unntak. Ideen er å sentralisere unntakshåndtering og forhindre at unntak blir stille slukt, samt åpne muligheter for Intelligent feilhåndtering.
2. **Gjentakelsesmekanismer**: Gjentakelsesmekanismer bidrar til å forbedre arbeidsflytens robusthet og håndtere midlertidige feil på en elegant måte. Du bør definitivt implementere gjentakelsesmekanismer for forbigående eller gjenopprettbare unntak, som problemer med nettverkstilkobling eller utilgjengelige ressurser som automatisk kan prøves på nytt etter en angitt

forsinkelse. Å ha en AI-drevet orkestrator eller unntakshåndterer betyr at gjentakelsesstrategiene dine ikke trenger å være mekaniske av natur, og være avhengige av faste algoritmer som eksponentiell tilbakestilling. Du kan overlate håndteringen av gjentakelsesforsøk til AI-komponentens "skjønn" som er ansvarlig for å bestemme hvordan unntaket skal håndteres.

3. **Reserveløsninger**: Hvis en AI-komponent ikke klarer å gi et gyldig svar eller støter på en feil—noe som er vanlig gitt dens banebrytende natur—ha en reservemekanisme på plass for å sikre at arbeidsflyten kan fortsette. Dette kan innebære bruk av standardverdier, alternative algoritmer, eller en Menneske i løkken for å ta beslutninger og holde arbeidsflyten i gang.
4. **Kompenserende handlinger**: Orkestratordirektivene bør inkludere instruksjoner om kompenserende handlinger for å håndtere unntak som ikke kan løses automatisk. Kompenserende handlinger er trinn som tas for å angre eller redusere effektene av en mislykket operasjon. For eksempel, hvis et betalingsprosesseringstrinn mislykkes, kan en kompenserende handling være å tilbakeføre transaksjonen og varsle brukeren. Kompenserende handlinger bidrar til å opprettholde datakonsistens og integritet når unntak oppstår.
5. **Unntaksovervåking og varsling**: Sett opp overvåkings- og varslingsmekanismer for å oppdage og varsle relevante interessenter om kritiske unntak. Orkestratoren kan gjøres oppmerksom på terskler og regler for å utløse varsler når unntak overskrider visse grenser eller når spesifikke typer unntak oppstår. Dette muliggjør proaktiv identifisering og løsning av problemer før de påvirker det overordnede systemet.

Her er et eksempel på unntakshåndtering og gjenoppretting i en Ruby-arbeidsflytkomponent:

```
class InventoryManager
  def check_availability(order)
    begin
      # Perform inventory check logic
      inventory = Inventory.find_by(product_id: order.product_id)
      if inventory.available_quantity >= order.quantity
        return true
      else
        raise InsufficientInventoryError,
              "Insufficient inventory for product #{order.product_id}"
      end
    rescue InsufficientInventoryError => e
      # Log the exception
      logger.error("Inventory check failed: #{e.message}")

      # Retry the operation after a delay
      retry_count ||= 0
      if retry_count < MAX_RETRIES
        retry_count += 1
        sleep(RETRY_DELAY)
        retry
      else
        # Fallback to manual intervention
        NotificationService.admin("Inventory check failed: Order #{order.id}")
        return false
      end
    end
  end
end
```

I dette eksempelet kontrollerer `InventoryManager`-komponenten tilgjengeligheten av et produkt for en gitt ordre. Hvis tilgjengelig mengde er utilstrekkelig, utløser den en `InsufficientInventoryError`. Unntaket fanges opp, logges, og en ny forsøksmekanisme implementeres. Hvis grensen for nye forsøk overskrides, faller komponenten tilbake på manuell intervensjon ved å varsle en administrator.

Ved å implementere robust unntakshåndtering og gjenopprettingsmekanismer, kan du sikre at dine intelligente arbeidsflyter er robuste, vedlikeholdbare og i stand til å

håndtere uventede situasjoner på en elegant måte.

Disse mønstrene danner grunnlaget for intelligent arbeidsflytorkestrering og kan kombineres og tilpasses for å møte de spesifikke kravene til ulike applikasjoner. Ved å utnytte disse mønstrene kan utviklere skape arbeidsflyter som er fleksible, robuste og optimalisert for ytelse og brukeropplevelse.

I neste del skal vi utforske hvordan disse mønstrene kan implementeres i praksis, ved å bruke eksempler fra virkeligheten og kodeutdrag for å illustrere integrasjonen av AI-komponenter i arbeidsflythåndtering.

Implementering av Intelligent Arbeidsflytorkestrering i Praksis

Nå som vi har utforsket nøkkelmønstrene i intelligent arbeidsflytorkestrering, la oss fordype oss i hvordan disse mønstrene kan implementeres i virkelige applikasjoner. Vi vil gi praktiske eksempler og kodeutdrag for å illustrere integrasjonen av AI-komponenter i arbeidsflythåndtering.

Intelligent Ordrebehandler

La oss se på et praktisk eksempel på implementering av intelligent arbeidsflytorkestrering ved å bruke en AI-drevet `OrderProcessor`-komponent i en Ruby on Rails e-handelsapplikasjon. `OrderProcessor` realiserer konseptet Process Manager Enterprise Integration som vi først møtte i Kapittel 3 da vi diskuterte Multitude of Workers. Komponenten vil være ansvarlig for å administrere ordrebehandlingsarbeidsflyten, ta rutingbeslutninger basert på mellomliggende resultater, og orkestrere utførelsen av ulike behandlingstrinn.

Ordrebehandlingsprosessen involverer flere trinn som ordrevalidering, lagerkontroll, betalingsbehandling og forsendelse. Hvert trinn er implementert som en separat arbeidsprosess som utfører en spesifikk oppgave og returnerer resultatet til `OrderProcessor`. Trinnene er ikke obligatoriske, og trenger ikke engang nødvendigvis å utføres i en bestemt rekkefølge.

Her er et eksempel på implementering av `OrderProcessor`. Den har to mixins fra Raix. Den første (`ChatCompletion`) gir den mulighet til å gjøre chatfullføring, som er det som gjør dette til en AI-komponent. Den andre (`FunctionDispatch`) muliggjør funksjonsanrop av AI-en, slik at den kan svare på en prompt med et funksjonsanrop i stedet for en tekstmelding.

Arbeiderfunksjonene (`validate_order`, `check_inventory`, et al) delegerer til sine respektive arbeiderklasser, som kan være AI- eller ikke-AI-komponenter, med det eneste kravet at de returnerer resultatene av arbeidet sitt i et format som kan representeres som en streng.

Som med alle andre eksempler i denne delen av boken, er denne koden praktisk talt pseudokode og er bare ment å formidle betydningen av mønsteret og inspirere dine egne kreasjoner. Fullstendige beskrivelser av mønstre og komplette kodeeksempler er inkludert i Del 2.

```ruby
class OrderProcessor
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are an order processor, tasked with..."

  def initialize(order)
    self.order = order
    transcript << { system: SYSTEM_DIRECTIVE }
    transcript << { user: order.to_json }
  end

  def perform
    # will continue looping until `stop_looping!` is called
    chat_completion(loop: true)
  end

  # list of functions available to be called by the AI
  # truncated for brevity

  def functions
    [
      {
        name: "validate_order",
        description: "Invoke to check validity of order",
        parameters: {
          ...
      },
      ...
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def validate_order
    OrderValidationWorker.perform(@order)
  end

  def check_inventory
    InventoryCheckWorker.perform(@order)
  end
```

```
43
44    def process_payment
45      PaymentProcessingWorker.perform(@order)
46    end
47
48    def schedule_shipping
49      ShippingSchedulerWorker.perform(@order)
50    end
51
52    def send_confirmation
53      OrderConfirmationWorker.perform(@order)
54    end
55
56    def finished_processing
57      @order.update!(transcript:, processed_at: Time.current)
58      stop_looping!
59    end
60  end
```

I eksempelet initialiseres OrderProcessor med et ordreobjekt og vedlikeholder en transkripsjon av arbeidsflyten, i det typiske samtaletranskripsjonsformatet som er naturlig for store språkmodeller. AI-en får full kontroll over orkestreringen av de forskjellige behandlingstrinnene, som ordrevalidering, lagerkontroll, betalingsbehandling og forsendelse.

Hver gang `chat_completion`-metoden kalles, sendes transkripsjonen til AI-en for å gi en fullføring som et funksjonsanrop. Det er helt opp til AI-en å analysere resultatet fra forrige trinn og bestemme passende handling. For eksempel, hvis lagerkontrollen avslører lave lagernivåer, kan `OrderProcessor` planlegge en etterfyllingsoppgave. Hvis betalingsbehandlingen mislykkes, kan den starte en ny forsøk eller varsle kundeservice.

> Eksempelet ovenfor har ikke definerte funksjoner for etterfylling eller varsling av kundeservice, men det kunne det absolutt hatt.

Transkripsjonen vokser hver gang en funksjon kalles og fungerer som en logg over arbeidsflyten, inkludert resultatene fra hvert trinn og AI-genererte instruksjoner for de neste trinnene. Denne transkripsjonen kan brukes til feilsøking, revisjon og for å gi innsyn i ordreoppfyllelsesprosessen.

Ved å utnytte AI i `OrderProcessor` kan e-handelsapplikasjonen dynamisk tilpasse arbeidsflyten basert på sanntidsdata og håndtere unntak på en intelligent måte. AI-komponenten kan ta informerte beslutninger, optimalisere arbeidsflyten og sikre smidig ordrebehandling selv i komplekse scenarioer.

Det faktum at det eneste kravet til arbeidsprosessene er å returnere en forståelig output som AI-en kan vurdere når den bestemmer neste handling, kan få deg til å innse hvordan denne tilnærmingen kan redusere inndata/utdata-kartleggingsarbeidet som vanligvis er involvert når man integrerer ulike systemer med hverandre.

Intelligent innholdsmoderering

Sosiale medier-applikasjoner krever generelt minimum innholdsmoderering for å sikre et trygt og sunt fellesskap. Dette eksempelet på en `ContentModerator`-komponent utnytter AI for intelligent orkestrering av modereringsprosessen, og tar beslutninger basert på innholdets egenskaper og resultatene fra ulike modereringstrinn.

Modereringsprosessen involverer flere trinn som tekstanalyse, bildegjenkjenning, vurdering av brukerrenommé og manuell gjennomgang. Hvert trinn er implementert som en separat arbeidsprosess som utfører en spesifikk oppgave og returnerer resultatet til `ContentModerator`.

Her er et eksempel på implementering av `ContentModerator`:

```ruby
class ContentModerator
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a content moderator process manager,
    tasked with the workflow involved in moderating user-generated content..."

  def initialize(content)
    @content = content
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: content.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  # list of functions available to be called by the AI
  # truncated for brevity

  def functions
    [
      {
        name: "analyze_text",
        # ...
      },
      {
        name: "recognize_image",
        description: "Invoke to describe images...",
        # ...
      },
      {
        name: "assess_user_reputation",
        # ...
      },
      {
```

```
        name: "escalate_to_manual_review",
        # ...
      },
      {
        name: "approve_content",
        # ...
      },
      {
        name: "reject_content",
        # ...
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def analyze_text
    result = TextAnalysisWorker.perform(@content)
    continue_with(result)
  end

  def recognize_image
    result = ImageRecognitionWorker.perform(@content)
    continue_with(result)
  end

  def assess_user_reputation
    result = UserReputationWorker.perform(@content.user)
    continue_with(result)
  end

  def escalate_to_manual_review
    ManualReviewWorker.perform(@content)
    @content.update!(status: 'pending', transcript: @transcript)
  end

  def approve_content
    @content.update!(status: 'approved', transcript: @transcript)
  end

  def reject_content
```

```
    @content.update!(status: 'rejected', transcript: @transcript)
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end
end
```

I dette eksemplet initialiseres `ContentModerator` med et innholdsobjekt og vedlikeholder en modereringslogg i samtaleformat. AI-komponenten har full kontroll over modereringsprosessen og bestemmer hvilke trinn som skal utføres basert på innholdets egenskaper og resultatene fra hvert trinn.

De tilgjengelige arbeiderfunksjonene som AI-en kan påkalle inkluderer `analyze_text`, `recognize_image`, `assess_user_reputation`, og `escalate_to_manual_review`. Hver funksjon delegerer oppgaven til en tilsvarende arbeiderprosess (`TextAnalysisWorker`, `ImageRecognitionWorker`, osv.) og legger til resultatet i modereringsloggen, med unntak av eskaleringsfunksjonen som fungerer som en slutttilstand. Til slutt fungerer også funksjonene `approve_content` og `reject_content` som slutttilstander.

AI-komponenten analyserer innholdet og bestemmer hvilken handling som skal tas. Hvis innholdet inneholder bildereferanser, kan den kalle på `recognize_image`-arbederen for å få hjelp med en visuell gjennomgang. Hvis noen arbeider advarer om potensielt skadelig innhold, kan AI-en bestemme seg for å eskalere innholdet til manuell gjennomgang eller rett og slett avvise det. Men avhengig av alvorlighetsgraden i advarselen, kan AI-en velge å bruke resultatene fra brukeromdømmevurderingen når den skal bestemme hvordan den skal håndtere innhold den ellers er usikker på. Avhengig av brukstilfellet kan det hende at betrodde brukere har mer spillerom i hva de kan publisere. Og så videre...

Som med det forrige prosessadministratoreksemplet fungerer modereringsloggen som

en oversikt over arbeidsflytens utførelse, inkludert resultatene fra hvert trinn og AI-genererte beslutninger. Denne loggen kan brukes til revisjon, åpenhet og forbedring av modereringsprosessen over tid.

Ved å utnytte AI i `ContentModerator` kan sosiale medier-applikasjonen dynamisk tilpasse modereringsarbeidsflyten basert på innholdets egenskaper og håndtere komplekse modereringsscenarier på en intelligent måte. AI-komponenten kan ta informerte beslutninger, optimalisere arbeidsflyten og sikre en trygg og sunn fellesskapsopplevelse.

La oss utforske to eksempler til som demonstrerer prediktiv oppgaveplanlegging og unntakshåndtering og gjenoppretting innenfor konteksten av intelligent arbeidsflytorkestrering.

Prediktiv oppgaveplanlegging i et kundeservicesystem

I en kundeserviceapplikasjon bygget med Ruby on Rails er effektiv håndtering og prioritering av støttehenvendelser avgjørende for å gi rettidig hjelp til kunder. `SupportTicketScheduler`-komponenten utnytter AI til prediktivt å planlegge og tildele støttehenvendelser til tilgjengelige agenter basert på ulike faktorer som henvendelsens hastegrad, agentens ekspertise og arbeidsbelastning.

```
class SupportTicketScheduler
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a support ticket scheduler,
    tasked with intelligently assigning tickets to available agents..."

  def initialize(ticket)
    @ticket = ticket
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: ticket.to_json }
    ]
```

```
14    end
15
16    def perform
17      complete(@transcript)
18    end
19
20    def model
21      "openai/gpt-4"
22    end
23
24    def functions
25      [
26        {
27          name: "analyze_ticket_urgency",
28          # ...
29        },
30        {
31          name: "list_available_agents",
32          description: "Includes expertise of available agents",
33          # ...
34        },
35        {
36          name: "predict_agent_workload",
37          description: "Uses historical data to predict upcoming workloads",
38          # ...
39        },
40        {
41          name: "assign_ticket_to_agent",
42          # ...
43        },
44        {
45          name: "reschedule_ticket",
46          # ...
47        }
48      ]
49    end
50
51    # implementation of functions that can be called by the AI
52    # entirely at its discretion, depending on the needs of the order
53
54    def analyze_ticket_urgency
55      result = TicketUrgencyAnalyzer.perform(@ticket)
```

```
    continue_with(result)
  end

  def list_available_agents
    result = ListAvailableAgents.perform
    continue_with(result)
  end

  def predict_agent_workload
    result = AgentWorkloadPredictor.perform
    continue_with(result)
  end

  def assign_ticket_to_agent
    TicketAssigner.perform(@ticket, @transcript)
  end

  def delay_assignment(until)
    until = DateTimeStandardizer.process(until)
    SupportTicketScheduler.delay(@ticket, @transcript, until)
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end
end
```

I dette eksempelet blir `SupportTicketScheduler` initialisert med et supporthenvendelsesobjekt og vedlikeholder en planleggingslogg. AI-komponenten analyserer henvendelsesdetaljene og planlegger prediktivt tildelingen av henvendelsen basert på faktorer som henvendelsens hastegrad, kundebehandlerens kompetanse og forventet arbeidsbelastning.

De tilgjengelige funksjonene som AI-en kan påkalle inkluderer `analyze_ticket_urgency`, `list_available_agents`, `predict_agent_workload`, og `assign_ticket_to_agent`. Hver funksjon delegerer oppgaven til en tilsvarende

analyse- eller prediktorkomponent og legger til resultatet i planleggingsloggen. AI-en har også muligheten til å utsette tildeling ved å bruke `delay_assignment`-funksjonen.

AI-komponenten undersøker planleggingsloggen og tar informerte beslutninger om henvendelsestildeling. Den vurderer henvendelsens hastegrad, tilgjengelige kundebehandleres kompetanse og den forventede arbeidsbelastningen for hver kundebehandler for å bestemme den best egnede kundebehandleren til å håndtere henvendelsen.

Ved å utnytte prediktiv oppgaveplanlegging kan kundesupportapplikasjonen optimalisere henvendelsestildeling, redusere responstider og forbedre den generelle kundetilfredsheten. Proaktiv og effektiv håndtering av supporthenvendelser sikrer at de riktige henvendelsene blir tildelt de riktige kundebehandlerne til rett tid.

Unntakshåndtering og gjenoppretting i en databehandlingspipeline

Håndtering av unntak og gjenoppretting etter feil er essensielt for å sikre dataintegritet og forhindre datatap.. `DataProcessingOrchestrator`-komponenten bruker AI til å intelligent håndtere unntak og orkestrere gjenopprettingsprosessen i en databehandlingspipeline

```
class DataProcessingOrchestrator
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a data processing orchestrator..."

  def initialize(data_batch)
    @data_batch = data_batch
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: data_batch.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  def functions
    [
      {
        name: "validate_data",
        # ...
      },
      {
        name: "process_data",
        # ...
      },
      {
        name: "request_fix",
        # ...
      },
      {
        name: "retry_processing",
        # ...
      },
      {
        name: "mark_data_as_failed",
```

```
        # ...
      },
      {
        name: "finished",
        # ...
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def validate_data
    result = DataValidator.perform(@data_batch)
    continue_with(result)
  rescue ValidationException => e
    handle_validation_exception(e)
  end

  def process_data
    result = DataProcessor.perform(@data_batch)
    continue_with(result)
  rescue ProcessingException => e
    handle_processing_exception(e)
  end

  def request_fix(description_of_fix)
    result = SmartDataFixer.new(description_of_fix, @data_batch)
    continue_with(result)
  end

  def retry_processing(timeout_in_seconds)
    wait(timeout_in_seconds)
    process_data
  end

  def mark_data_as_failed
    @data_batch.update!(status: 'failed', transcript: @transcript)
  end

  def finished
    @data_batch.update!(status: 'finished', transcript: @transcript)
```

```
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end

  def handle_validation_exception(exception)
    @transcript << { exception: exception.message }
    complete(@transcript)
  end

  def handle_processing_exception(exception)
    @transcript << { exception: exception.message }
    complete(@transcript)
  end
end
```

I dette eksempelet initialiseres `DataProcessingOrchestrator` med et databatch-objekt og vedlikeholder en behandlingslogg. AI-komponenten orkestrerer databehandlingsprosessen, håndterer unntak og gjenoppretter fra feil etter behov.

Funksjonene som er tilgjengelige for AI-en å påkalle inkluderer `validate_data`, `process_data`, `request_fix`, `retry_processing`, og `mark_data_as_failed`. Hver funksjon delegerer oppgaven til en tilsvarende databehandlingskomponent og legger til resultatet eller unntaksdetaljene i behandlingsloggen.

Hvis et valideringsunntak oppstår under `validate_data`-steget, vil `handle_validation_exception`-funksjonen legge til unntaksdataene i loggen og gi kontrollen tilbake til AI-en. På samme måte, hvis et behandlingsunntak oppstår under `process_data`-steget, kan AI-en bestemme gjenopprettingsstrategien.

Avhengig av typen unntak som oppstår, kan AI-en etter eget skjønn velge å kalle `request_fix`, som delegerer til en AI-drevet `SmartDataFixer`-komponent (se kapittelet om Selvhelbredende Data). Datafiksereren får en enkel beskrivelse på engelsk

om hvordan den skal modifisere `@data_batch` slik at behandlingen kan prøves på nytt. Kanskje en vellykket ny behandling ville innebære å fjerne poster fra databatchen som ikke har bestått validering og/eller kopiere dem til en annen behandlingsprosess for manuell gjennomgang? Mulighetene er nesten uendelige.

Ved å inkorporere AI-drevet unntakshåndtering og gjenoppretting blir databehandlingsapplikasjonen mer robust og feiltolerant. `DataProcessingOrchestrator` håndterer unntak intelligent, minimerer datatap og sikrer en smidig gjennomføring av databehandlingsarbeidsflyten.

Overvåking og Logging

Overvåking og logging gir innsikt i fremgangen, ytelsen og tilstanden til AI-drevne arbeidsflytkomponenter, og gjør det mulig for utviklere å spore og analysere systemets oppførsel. Implementering av effektive overvåkings- og loggingsmekanismer er essensielt for feilsøking, revisjon og kontinuerlig forbedring av intelligente arbeidsflyter.

Overvåking av Arbeidsflytens Fremgang og Ytelse

For å sikre en smidig gjennomføring av intelligente arbeidsflyter er det viktig å overvåke fremgangen og ytelsen til hver arbeidsflytkomponent. Dette innebærer å spore viktige målinger og hendelser gjennom arbeidsflytens livssyklus.

Noen viktige aspekter å overvåke inkluderer:

1. **Arbeidsflytens Kjøretid**: Mål tiden hver arbeidsflytkomponent bruker på å fullføre sin oppgave. Dette hjelper med å identifisere ytelsesproblemer og optimalisere den generelle arbeidsflyteffektiviteten.

2. **Ressursutnyttelse**: Overvåk bruken av systemressurser, som CPU, minne og lagring, for hver arbeidsflytkomponent. Dette hjelper med å sikre at systemet opererer innenfor sin kapasitet og kan håndtere arbeidsmengden effektivt.

3. **Feilrater og Unntak**: Spor forekomsten av feil og unntak innenfor arbeidsflytkomponenter. Dette hjelper med å identifisere potensielle problemer og muliggjør proaktiv feilhåndtering og gjenoppretting.

4. **Beslutningspunkter og Utfall**: Overvåk beslutningspunktene i arbeidsflyten og utfallene av AI-drevne beslutninger. Dette gir innsikt i oppførselen og effektiviteten til AI-komponentene.

Dataene som fanges opp av overvåkingsprosessene kan vises i dashbord eller brukes som input til planlagte rapporter som informerer systemadministratorer om systemets tilstand.

Overvåkingsdata kan mates til en AI-drevet systemadministratorprosess for gjennomgang og potensielle tiltak!

Logging av Viktige Hendelser og Beslutninger

Logging er en essensiell praksis som innebærer å fange og lagre relevant informasjon om viktige hendelser, beslutninger og unntak som oppstår under arbeidsflytens utførelse.

Noen viktige aspekter å logge inkluderer:

1. **Arbeidsflytinitiering og Fullføring**: Logg start- og sluttidspunkter for hver arbeidsflytinstans, sammen med relevant metadata som inputdata og brukerkontekst.

2. **Komponentutførelse**: Logg utførelsesdetaljene for hver arbeidsflytkomponent, inkludert inputparametere, outputresultater og eventuelle mellomliggende data som genereres.

3. **AI-beslutninger og Resonnement**: Logg beslutningene tatt av AI-komponenter, sammen med det underliggende resonnementet eller konfidensscorer. Dette gir transparens og muliggjør revisjon av AI-drevne beslutninger.

4. **Unntak og Feilmeldinger**: Logg eventuelle unntak eller feilmeldinger som oppstår under arbeidsflytutførelsen, inkludert stabelsporing og relevant kontekstinformasjon.

Logging kan implementeres ved hjelp av ulike teknikker, som å skrive til loggfiler, lagre logger i en database eller sende logger til en sentralisert loggtjeneste. Det er viktig å velge et loggingsrammeverk som gir fleksibilitet, skalerbarhet og enkel integrasjon med applikasjonens arkitektur.

Her er et eksempel på hvordan logging kan implementeres i en Ruby on Rails-applikasjon ved hjelp av `ActiveSupport::Logger`-klassen:

```
class WorkflowLogger
  def self.log(message, severity = :info)
    @logger ||= ActiveSupport::Logger.new('workflow.log')
    @logger.formatter ||= proc do |severity, datetime, progname, msg|
      "#{datetime} [#{severity}] #{msg}\n"
    end
    @logger.send(severity, message)
  end
end

# Usage example
WorkflowLogger.log("Workflow initiated for order ##{@order.id}")
WorkflowLogger.log("Payment processing completed successfully")
WorkflowLogger.log("Inventory check failed for item ##{item.id}", :error)
```

Ved å strategisk plassere loggføringserklæringer gjennom arbeidsflytkomponentene og AI-beslutningspunktene, kan utviklere fange opp verdifull informasjon for feilsøking, revisjon og analyse.

Fordeler med Overvåking og Logging

Implementering av overvåking og logging i intelligent arbeidsflytorkestrering gir flere fordeler:

1. **Feilsøking og Problemløsing**: Detaljerte logger og overvåkingsdata hjelper utviklere med å identifisere og diagnostisere problemer raskt. De gir innsikt i arbeidsflytens utførelsesflyt, komponentinteraksjoner og eventuelle feil eller unntak som oppstår.

2. **Ytelsesoptimalisering**: Overvåking av ytelsesmetrikker gjør det mulig for utviklere å identifisere flaskehalser og optimalisere arbeidsflytkomponentene for bedre effektivitet. Ved å analysere kjøretider, ressursbruk og andre metrikker, kan utviklere ta informerte beslutninger for å forbedre systemets generelle ytelse.

3. **Revisjon og Etterlevelse**: Logging av viktige hendelser og beslutninger gir et revisjonsspor for regulatorisk etterlevelse og ansvarlighet. Det gjør det mulig for organisasjoner å spore og verifisere handlingene utført av AI-komponenter og sikre overholdelse av forretningsregler og juridiske krav.

4. **Kontinuerlig Forbedring**: Overvåkings- og loggingsdata fungerer som verdifulle innspill for kontinuerlig forbedring av intelligente arbeidsflyter. Ved å analysere historiske data, identifisere mønstre og måle effektiviteten av AI-beslutninger, kan utviklere iterativt forbedre og styrke arbeidsflytorkestreringslogikken.

Hensyn og Beste Praksis

Ved implementering av overvåking og logging i intelligent arbeidsflytorkestrering, vurder følgende beste praksis:

1. **Definer Klare Overvåkingsmetrikker**: Identifiser de viktigste metrikkene og hendelsene som må overvåkes basert på arbeidsflytens spesifikke krav. Fokuser på metrikker som gir meningsfull innsikt i systemets ytelse, helse og oppførsel.

2. **Implementer Detaljert Logging**: Sørg for at loggføringserklæringer er plassert på passende punkter innenfor arbeidsflytkomponentene og AI-beslutningspunktene. Fang opp relevant kontekstinformasjon, som inngangsparametere, utgangsresultater og eventuelle mellomliggende data som genereres.

3. **Bruk Strukturert Logging**: Ta i bruk et strukturert loggformat for å forenkle parsing og analyse av loggdata. Strukturert logging muliggjør bedre søkbarhet, filtrering og aggregering av loggoppføringer.

4. **Administrer Loggoppbevaring og -rotasjon**: Implementer retningslinjer for loggoppbevaring og -rotasjon for å administrere lagring og livssyklus for loggfiler. Bestem passende oppbevaringsperiode basert på juridiske krav, lagringsbegrensninger og analysebehov. Hvis mulig, outsource logging til en tredjepartstjeneste som Papertrail.

5. **Sikre Sensitiv Informasjon**: Vær forsiktig ved logging av sensitiv informasjon, som personidentifiserbar informasjon (PII) eller konfidensiell forretningsdata. Implementer passende sikkerhetstiltak, som datamasking eller kryptering, for å beskytte sensitiv informasjon i loggfiler.

6. **Integrer med Overvåkings- og Varslingsverktøy**: Utnytt overvåkings- og varslingsverktøy for å sentralisere innsamling, analyse og visualisering av overvåkings- og loggdata. Disse verktøyene kan gi sanntidsinnsikt, generere varsler basert på forhåndsdefinerte terskler og legge til rette for proaktiv problemoppdagelse og -løsning. Mitt favorittverktøy blant disse er Datadog.

Ved å implementere omfattende overvåkings- og loggingsmekanismer, kan utviklere få verdifull innsikt i oppførselen og ytelsen til intelligente arbeidsflyter. Denne innsikten muliggjør effektiv feilsøking, optimalisering og kontinuerlig forbedring av AI-drevne arbeidsflytorkestreringssystemer.

Skalerbarhet og Ytelseshensyn

Skalerbarhet og ytelse er kritiske aspekter å vurdere når man designer og implementerer intelligente arbeidsflytorkestreringssystemer. Ettersom volumet av samtidige arbeidsflyter og kompleksiteten av AI-drevne komponenter øker, blir det essensielt å sikre at systemet kan håndtere arbeidsmengden effektivt og skalere sømløst for å møte voksende behov.

Håndtering av Store Volumer av Samtidige Arbeidsflyter

Intelligente arbeidsflytorkestreringssystemer må ofte håndtere et stort antall samtidige arbeidsflyter. For å sikre skalerbarhet, vurder følgende strategier:

1. **Asynkron Behandling**: Implementer asynkrone behandlingsmekanismer for å løskoble utførelsen av arbeidsflytkomponenter. Dette gjør det mulig for systemet å håndtere flere arbeidsflyter samtidig uten blokkering eller venting på at hver komponent skal fullføres. Asynkron behandling kan oppnås ved bruk av meldingskøer, hendelsesdrevne arkitekturer eller bakgrunnsjobbrammeverk som Sidekiq.

2. **Distribuert Arkitektur**: Design systemarkitekturen for å bruke serverløse komponenter (som AWS Lambda) eller enkelt distribuere arbeidsmengden på tvers av flere noder eller servere sammen med hovedapplikasjonsserveren. Dette muliggjør horisontal skalerbarhet, hvor flere noder kan legges til for å håndtere økte arbeidsflytvolumer.

3. **Parallell Utførelse**: Identifiser muligheter for parallell utførelse innenfor arbeidsflyter. Noen arbeidsflytkomponenter kan være uavhengige av hverandre og kan utføres samtidig. Ved å utnytte parallelle prosesseringsteknikker, som multitråding eller distribuerte oppgavekøer, kan systemet optimalisere ressursutnyttelsen og redusere total arbeidsflytutførelsestid.

Optimalisering av ytelsen til AI-drevne komponenter

AI-drevne komponenter, som maskinlæringsmodeller eller systemer for behandling av naturlig språk, kan være beregningsmessig krevende og påvirke den generelle ytelsen til arbeidsflytorkestreringssystemet. For å optimalisere ytelsen til AI-komponenter, vurder følgende teknikker:

1. **Hurtigbufring**: Hvis AI-behandlingen din er rent generativ og ikke involverer sanntidsinformasjonsoppslag eller eksterne integrasjoner for å generere chatfullføringer,

kan du undersøke hurtigbufringsmekanismer for å lagre og gjenbruke resultatene av hyppig brukte eller beregningsmessig kostbare operasjoner.

2. **Modelloptimalisering**: Optimaliser kontinuerlig måten du bruker AI-modeller i arbeidsflytkomponenter. Dette kan innebære teknikker som *Prompt-destillering* eller det kan rett og slett være et spørsmål om å teste nye modeller etter hvert som de blir tilgjengelige.

3. **Satsvis behandling**: Hvis du jobber med GPT-4-klasse modeller, kan du kanskje utnytte satsvise behandlingsteknikker for å behandle flere datapunkter eller forespørsler i én enkelt sats, i stedet for å behandle dem individuelt. Ved å behandle data i satser, kan systemet optimalisere ressursutnyttelsen og redusere overhead fra gjentatte modellforespørsler.

Overvåking og profilering av ytelse

For å identifisere ytelsesproblemer og optimalisere skalerbarheten til det intelligente arbeidsflytorkestreringssystemet, er det avgjørende å implementere overvåkings- og profileringsmekanismer. Vurder følgende tilnærminger:

1. **Ytelsesmålinger**: Definer og spor viktige ytelsesmålinger, som responstid, gjennomstrømning, ressursutnyttelse og forsinkelse. Disse målingene gir innsikt i systemets ytelse og hjelper med å identifisere områder for optimalisering. Den populære AI-modell-aggregatoren OpenRouter inkluderer Host[1]- og Speed[2]-målinger i hvert API-svar, noe som gjør det enkelt å spore disse viktige målingene.

2. **Profileringsverktøy**: Bruk profileringsverktøy for å analysere ytelsen til individuelle arbeidsflytkomponenter og AI-operasjoner. Profileringsverktøy kan hjelpe med å identifisere ytelsesflaskehalser, ineffektive kodestier eller ressurskrevende

[1] Host er tiden det tok å motta den første byten av den strømmede genereringen fra modellverten, også kjent som "tid til første byte."

[2] Speed beregnes som antall fullføringstokens delt på total genereringstid. For ikke-strømmede forespørsler regnes forsinkelse som en del av genereringstiden.

operasjoner. Populære profileringsverktøy inkluderer New Relic, Scout, eller innebygde profileringsverktøy som følger med programmeringsspråket eller rammeverket.

3. **Belastningstesting**: Gjennomfør belastningstesting for å evaluere systemets ytelse under forskjellige nivåer av samtidige arbeidsbelastninger. Belastningstesting hjelper med å identifisere systemets skaleringsgrenser, oppdage ytelsesforringelse og sikre at systemet kan håndtere forventet trafikk uten å kompromittere ytelsen.

4. **Kontinuerlig overvåking**: Implementer kontinuerlige overvåkings- og varslingsmekanismer for proaktivt å oppdage ytelsesproblemer og flaskehalser. Sett opp overvåkingsdashbord og varsler for å spore viktige ytelsesindikator (KPI-er) og motta varsler når forhåndsdefinerte terskler overskrides. Dette muliggjør rask identifisering og løsning av ytelsesproblemer.

Skaleringsstrategier

For å håndtere økende arbeidsbelastninger og sikre skalerbarheten til det intelligente arbeidsflytorkestreringssystemet, vurder følgende skaleringsstrategier:

1. **Vertikal skalering**: Vertikal skalering innebærer å øke ressursene (f.eks. CPU, minne) til individuelle noder eller servere for å håndtere høyere arbeidsbelastninger. Denne tilnærmingen er egnet når systemet krever mer prosesseringskraft eller minne for å håndtere komplekse arbeidsflyter eller AI-operasjoner.

2. **Horisontal skalering**: Horisontal skalering innebærer å legge til flere noder eller servere i systemet for å distribuere arbeidsbelastningen. Denne tilnærmingen er effektiv når systemet må håndtere et stort antall samtidige arbeidsflyter eller når arbeidsbelastningen enkelt kan distribueres over flere noder. Horisontal skalering krever en distribuert arkitektur og lastbalanseringsmekanismer for å sikre jevn distribusjon av trafikk.

3. **Automatisk skalering**: Implementer automatiske skaleringsmekanismer for automatisk å justere antall noder eller ressurser basert på arbeidsbelastningsbehovet.

Automatisk skalering lar systemet dynamisk skalere opp eller ned avhengig av innkommende trafikk, noe som sikrer optimal ressursutnyttelse og kostnadseffektivitet. Skyplattformer som Amazon Web Services (AWS) eller Google Cloud Platform (GCP) tilbyr automatiske skaleringsmuligheter som kan utnyttes for intelligente arbeidsflytorkestreringssystemer.

Ytelsesoptimaliseringsteknikker

I tillegg til skaleringsstrategiene, vurder følgende ytelsesoptimaliseringsteknikker for å forbedre effektiviteten til det intelligente arbeidsflytorkestreringssystemet:

1. **Effektiv datalagring og -henting**: Optimaliser mekanismene for datalagring og -henting som brukes av arbeidsflytkomponentene. Bruk effektiv databaseindeksering, spørringsoptimaliseringsteknikker og databufring for å minimere forsinkelsen og forbedre ytelsen til dataintensive operasjoner.

2. **Asynkron I/O**: Benytt asynkrone I/O-operasjoner for å forhindre blokkering og forbedre systemets responstid. Asynkron I/O gjør det mulig for systemet å håndtere flere forespørsler samtidig uten å måtte vente på at I/O-operasjoner skal fullføres, og dermed maksimere ressursutnyttelsen.

3. **Effektiv serialisering og deserialisering**: Optimaliser serialiserings- og deserialiseringsprosessene som brukes til datautveksling mellom arbeidsflytkomponenter. Bruk effektive serialiseringsformater som Protocol Buffers eller MessagePack for å redusere overhead ved dataserialisering og forbedre ytelsen i kommunikasjonen mellom komponenter.

For Ruby-baserte applikasjoner, vurder å bruke Universal ID. Universal ID utnytter både MessagePack og Brotli (en kombinasjon bygget for hastighet og førsteklasses datakomprimering). Når disse bibliotekene kombineres, er de opptil 30% raskere og oppnår komprimeringsrater som ligger innenfor 2-5% sammenlignet med Protocol Buffers.

4. **Komprimering og koding**: Bruk komprimerings- og kodingsteknikker for å redusere størrelsen på data som overføres mellom arbeidsflytkomponenter. Komprimeringsalgoritmer som gzip eller Brotli kan betydelig redusere bruken av nettverksbåndbredde og forbedre systemets generelle ytelse.

Ved å ta hensyn til skalerbarhet og ytelsesaspekter under design og implementering av intelligente arbeidsflytorkestreringssystemer, kan du sikre at systemet ditt kan håndtere store volumer av samtidige arbeidsflyter, optimalisere ytelsen til AI-drevne komponenter og skalere sømløst for å møte økende behov. Kontinuerlig overvåking, profilering og optimalisering er essensielt for å opprettholde systemets ytelse og responstid etter hvert som arbeidsmengden og kompleksiteten øker over tid.

Testing og validering av arbeidsflyter

Testing og validering er kritiske aspekter ved utvikling og vedlikehold av intelligente arbeidsflytorkestreringssystemer. Gitt den komplekse naturen til AI-drevne arbeidsflyter, er det essensielt å sikre at hver komponent fungerer som forventet, at den overordnede arbeidsflyten oppfører seg korrekt, og at AI-beslutningene er nøyaktige og pålitelige. I denne delen skal vi utforske ulike teknikker og hensyn for testing og validering av intelligente arbeidsflyter.

Enhetstesting av arbeidsflytkomponenter

Enhetstesting innebærer å teste individuelle arbeidsflytkomponenter isolert for å verifisere deres korrekthet og robusthet. Når man enhetstester AI-drevne arbeidsflytkomponenter, bør man vurdere følgende:

1. **Inputvalidering**: Test komponentens evne til å håndtere forskjellige typer input, inkludert gyldig og ugyldig data. Verifiser at komponenten håndterer kanttilfeller på en god måte og gir passende feilmeldinger eller unntak.

2. **Outputverifisering**: Bekreft at komponenten produserer forventet output for et gitt sett med input. Sammenlign faktisk output med forventede resultater for å sikre korrekthet.

3. **Feilhåndtering**: Test komponentens feilhåndteringsmekanismer ved å simulere ulike feilscenarier, som ugyldig input, utilgjengelige ressurser eller uventede unntak. Verifiser at komponenten fanger opp og håndterer feil på en hensiktsmessig måte.

4. **Grensebetingelser**: Test komponentens oppførsel under grensebetingelser, som tom input, maksimal inputstørrelse eller ekstreme verdier. Sørg for at komponenten håndterer disse betingelsene på en god måte uten å krasje eller produsere feil resultater.

Her er et eksempel på en enhetstest for en arbeidsflytkomponent i Ruby ved bruk av RSpec-testrammeverket:

```
RSpec.describe OrderValidator do
  describe '#validate' do
    context 'when order is valid' do
      let(:order) { build(:order) }

      it 'returns true' do
        expect(subject.validate(order)).to be true
      end
    end

    context 'when order is invalid' do
      let(:order) { build(:order, total_amount: -100) }

      it 'returns false' do
        expect(subject.validate(order)).to be false
      end
    end
  end
end
```

I dette eksempelet blir `OrderValidator`-komponenten testet ved hjelp av to testtilfeller: ett for en gyldig ordre og ett for en ugyldig ordre. Testtilfellene verifiserer

at `validate`-metoden returnerer den forventede boolske verdien basert på ordrens gyldighet.

Integrasjonstesting av Arbeidsflytinteraksjoner

Integrasjonstesting fokuserer på å verifisere interaksjonene og dataflyten mellom ulike arbeidsflytkomponenter. Det sikrer at komponentene fungerer sømløst sammen og produserer de forventede resultatene. Ved integrasjonstesting av intelligente arbeidsflyter bør du vurdere følgende:

1. **Komponentinteraksjon**: Test kommunikasjonen og datautvekslingen mellom arbeidsflytkomponenter. Verifiser at output fra én komponent blir korrekt overført som input til neste komponent i arbeidsflyten.

2. **Datakonsistens**: Sørg for at data forblir konsistente og nøyaktige mens de flyter gjennom arbeidsflyten. Verifiser at datatransformasjoner, beregninger og aggregeringer utføres korrekt.

3. **Unntakspropagering**: Test hvordan unntak og feil propageres og håndteres på tvers av arbeidsflytkomponenter. Verifiser at unntak fanges opp, logges og håndteres på en hensiktsmessig måte for å forhindre forstyrrelser i arbeidsflyten.

4. **Asynkron oppførsel**: Hvis arbeidsflyten involverer asynkrone komponenter eller parallell eksekvering, test koordinerings- og synkroniseringsmekanismene. Sørg for at arbeidsflyten oppfører seg korrekt under samtidige og asynkrone scenarier.

Her er et eksempel på en integrasjonstest for en arbeidsflyt i Ruby ved hjelp av RSpec-testrammeverket:

```ruby
RSpec.describe OrderProcessingWorkflow do

  let(:order) { build(:order) }

  it 'processes the order successfully' do
    expect(OrderValidator).to receive(:validate).and_return(true)
    expect(InventoryManager).to receive(:check_availability).and_return(true)
    expect(PaymentProcessor).to receive(:process_payment).and_return(true)
    expect(ShippingService).to receive(:schedule_shipping).and_return(true)

    workflow = OrderProcessingWorkflow.new(order)
    result = workflow.process

    expect(result).to be true
    expect(order.status).to eq('processed')
  end

end
```

I dette eksempelet testes `OrderProcessingWorkflow` ved å verifisere samhandlingen mellom ulike arbeidsflytkomponenter. Testtilfellet setter opp forventninger til hver komponents oppførsel og sikrer at arbeidsflyten behandler ordren vellykket, med tilhørende oppdatering av ordrens status.

Testing av AI-beslutningspunkter

Testing av AI-beslutningspunkter er avgjørende for å sikre nøyaktighet og pålitelighet i AI-drevne arbeidsflyter. Ved testing av AI-beslutningspunkter bør du vurdere følgende:

1. **Beslutningsnøyaktighet**: Verifiser at AI-komponenten tar nøyaktige beslutninger basert på inndata og den trente modellen. Sammenlign AI-beslutningene med forventede resultater eller referansedata.

2. **Kanttilfeller**: Test AI-komponentens oppførsel under kanttilfeller og uvanlige scenarioer. Verifiser at AI-komponenten håndterer disse tilfellene på en god måte og tar fornuftige beslutninger.

3. **Skjevhet og rettferdighet**: Vurder AI-komponenten for potensielle skjevheter og sørg for at den tar rettferdige og upartiske beslutninger. Test komponenten med varierte inndata og analyser resultatene for å identifisere eventuelle diskriminerende mønstre.

4. **Forklarbarhet**: Hvis AI-komponenten gir forklaringer eller resonnementer for sine beslutninger, verifiser at forklaringene er korrekte og tydelige. Sørg for at forklaringene samsvarer med den underliggende beslutningsprosessen.

Her er et eksempel på testing av et AI-beslutningspunkt i Ruby ved bruk av RSpec-testrammeverket:

```
RSpec.describe FraudDetector do
  describe '#detect_fraud' do
    context 'when transaction is fraudulent' do
      let(:tx) do
        build(:transaction, amount: 10_000, location: 'High-Risk Country')
      end

      it 'returns true' do
        expect(subject.detect_fraud(tx)).to be true
      end
    end

    context 'when transaction is legitimate' do
      let(:tx) do
        build(:transaction, amount: 100, location: 'Low-Risk Country')
      end

      it 'returns false' do
        expect(subject.detect_fraud(tx)).to be false
      end
    end
  end
end
```

I dette eksemplet blir `FraudDetector` AI-komponenten testet med to testtilfeller: ett for en svindelaktig transaksjon og ett annet for en legitim transaksjon. Testtilfellene verifiserer at `detect_fraud`-metoden returnerer den forventede boolske verdien basert på transaksjonens egenskaper.

Ende-til-ende-testing

Ende-til-ende-testing innebærer å teste hele arbeidsflyten fra start til slutt, simulere virkelige scenarioer og brukerinteraksjoner. Det sikrer at arbeidsflyten oppfører seg korrekt og produserer de ønskede resultatene. Når man utfører ende-til-ende-testing for intelligente arbeidsflyter, bør man vurdere følgende:

1. **Brukerscenarioer**: Identifiser vanlige brukerscenarioer og test arbeidsflytens oppførsel under disse scenarioene. Verifiser at arbeidsflyten håndterer brukerinndata korrekt, tar passende beslutninger og produserer de forventede resultatene.

2. **Datavalidering**: Sørg for at arbeidsflyten validerer og renser brukerinndata for å forhindre datauoverensstemmelser eller sikkerhetssårbarheter. Test arbeidsflyten med forskjellige typer inndata, inkludert både gyldige og ugyldige data.

3. **Feilhåndtering**: Test arbeidsflytens evne til å gjenopprette seg fra feil og unntak. Simuler feilscenarioer og verifiser at arbeidsflyten håndterer dem på en elegant måte, logger feilene og tar passende gjenopprettingstiltak.

4. **Ytelse og Skalerbarhet**: Vurder arbeidsflytens ytelse og skalerbarhet under forskjellige lastforhold. Test arbeidsflyten med et stort volum av samtidige forespørsler og mål responstider, ressursbruk og generell systemstabilitet.

Her er et eksempel på en ende-til-ende-test for en arbeidsflyt i Ruby ved hjelp av RSpec testrammeverket og Capybara-biblioteket for å simulere brukerinteraksjoner:

```
RSpec.describe 'Order Processing Workflow' do
  scenario 'User places an order successfully' do
    visit '/orders/new'
    fill_in 'Product', with: 'Sample Product'
    fill_in 'Quantity', with: '2'
    fill_in 'Shipping Address', with: '123 Main St'
    click_button 'Place Order'

    expect(page).to have_content('Order Placed Successfully')
    expect(Order.count).to eq(1)
    expect(Order.last.status).to eq('processed')
  end
end
```

I dette eksempelet simulerer ende-til-ende-testen en bruker som legger inn en bestilling gjennom nettgrensesnittet. Den fyller ut de nødvendige skjemafeltene, sender inn bestillingen og bekrefter at bestillingen blir behandlet vellykket, viser den riktige bekreftelsesmeldingen og oppdaterer bestillingens status i databasen.

Kontinuerlig integrasjon og distribusjon

For å sikre pålitelighet og vedlikeholdbarhet i intelligente arbeidsflyter, anbefales det å integrere testing og validering i den kontinuerlige integrasjons- og distribusjons (CI/CD)-pipeline. Dette muliggjør automatisert testing og validering av endringer i arbeidsflyten før de distribueres til produksjon. Vurder følgende praksis:

1. **Automatisert testkjøring**: Konfigurer CI/CD-pipelinen til å kjøre testsuiten automatisk når det gjøres endringer i arbeidsflytens kodebase. Dette sikrer at eventuelle regresjoner eller feil oppdages tidlig i utviklingsprosessen.

2. **Overvåking av testdekning**: Mål og overvåk testdekningen av arbeidsflytkomponentene og AI-beslutningspunktene. Sikt mot høy testdekning for å sikre at kritiske baner og scenarioer blir grundig testet.

3. **Kontinuerlig tilbakemelding**: Integrer testresultater og kodekvalitetsmetrikker i utviklingsarbeidsflyten. Gi kontinuerlig tilbakemelding til utviklere om status på tester,

kodekvalitet og eventuelle problemer som oppdages under CI/CD-prosessen.

4. **Testmiljøer**: Distribuer arbeidsflyten til testmiljøer som ligger tett opptil produksjonsmiljøet. Utfør ytterligere testing og validering i testmiljøet for å fange opp eventuelle problemer relatert til infrastruktur, konfigurasjon eller dataintegrasjon.

5. **Tilbakerullingsmekanismer**: Implementer tilbakerullingsmekanismer i tilfelle distribueringsfeil eller kritiske problemer oppdages i produksjon. Sørg for at arbeidsflyten raskt kan rulles tilbake til en tidligere stabil versjon for å minimere nedetid og påvirkning på brukerne.

Ved å inkorporere testing og validering gjennom hele utviklingslivssyklusen til intelligente arbeidsflyter, kan organisasjoner sikre pålitelighet, nøyaktighet og vedlikeholdbarhet i deres AI-drevne systemer. Regelmessig testing og validering hjelper med å fange opp feil, forhindre regresjoner og bygge tillit til arbeidsflytens oppførsel og resultater.

Del 2: Mønstrene

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Prompt-konstruksjon

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Tankerekke

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Innholdsgenerering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Strukturert entitetsopprettelse

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Veiledning av LLM-agenter

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Fordeler og hensyn

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Modusveksling

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Når den skal brukes

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Rolletildeling

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Når det skal brukes

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Prompt Object

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Prompt Template

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Fordeler og hensyn

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Når det bør brukes:

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Structured IO

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Skalering av Structured IO

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Fordeler og hensyn

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Prompt-kjeding

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Når du bør bruke det

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel: Olympias Onboarding

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Prompt-omskriver

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Responsbegrensning

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Fordeler og hensyn

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Feilhåndtering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Spørringsanalysator

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Implementering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Ordklassemerking (POS) og navngitt enhetgjenkjenning (NER)

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Intensjonsklassifisering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Nøkkelorduttrekking

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Fordeler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Spørringsomskriver

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Fordeler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Ventriloquist

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Når man skal bruke det

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Diskrete komponenter

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Predikat

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Når det skal brukes

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

API-fasade

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hovedfordeler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Når man skal bruke det

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Autentisering og autorisering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Håndtering av forespørsler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Formatering av respons

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Feilhåndtering og kanttilfeller

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Skalerbarhets- og ytelseshensyn

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Sammenligning med andre designmønstre

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Result Interpreter

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Når det skal brukes

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Virtuell Maskin

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Når det skal brukes

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Bak Magien

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Spesifikasjon og Testing

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Spesifisering av Oppførsel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Skriving av Testtilfeller

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel: Testing av Oversetterkomponenten

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Replay av HTTP-interaksjoner

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Human In The Loop (HITL)

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Overordnede mønstre

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hybrid intelligens

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Adaptiv respons

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Menneske-KI-rollebytte

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eskalering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hovedfordeler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Praktisk anvendelse: Helsevesen

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Tilbakemeldingssløyfe

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Anvendelser og Eksempler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Avanserte Teknikker i Integrering av Menneskelig Tilbakemelding

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Passiv informasjonsutstråling

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Kontekstuell informasjonsvisning

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Proaktive varsler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Forklarende innsikt

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Interaktiv utforskning

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Viktige fordeler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Bruksområder og eksempler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Samarbeidende Beslutningstaking (CDM)

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Kontinuerlig læring

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Bruksområder og eksempler

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Etiske hensyn

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

HITLs rolle i reduksjon av AI-risiko

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Teknologiske fremskritt og fremtidsutsikter

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Utfordringer og begrensninger ved HITL-systemer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Intelligent feilhåndtering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Tradisjonelle feilhåndteringstilnærminger

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Kontekstuell feildiagnose

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Prompt-konstruksjon for kontekstuell feildiagnose

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Gjenfinningsforsterket generering for kontekstuell feildiagnose

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Intelligent feilrapportering

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Prediktiv feilforebygging

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Smart feilgjenoppretting

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Personalisert feilkommunikasjon

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Adaptiv feilhåndteringsarbeidsflyt

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Kvalitetskontroll

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eval

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Problem

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Løsning

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hensyn

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Forståelse av gullstandarder

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan referansefrie evalueringer fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Sikkerhetsmekanisme

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Problem

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Løsning

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hvordan det fungerer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Eksempel

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Hensyn å ta

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Guardrails og Evalueringer: To Sider av Samme Sak

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Utbyttbarheten mellom Guardrails og Referansefrie Evalueringer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Implementering av Tosidige Sikkerhetsmekanismer og Evalueringer

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Ordliste

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Ordliste

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

A

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

B

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

C

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

D

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

E

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

F

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

G

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

H

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

I

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

J

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

K

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

L

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

M

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

N

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

O

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

P

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Q

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

R

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

S

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

T

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

U

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

V

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

W

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Z

Dette innholdet er ikke tilgjengelig i prøveboken. Boken kan kjøpes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-nb.

Index

ACID properties, 103
adaptiv arbeidsflyt
    Adaptiv arbeidsflytkomposisjon, 213
adaptivt UI, 196
Agentiske, 29
AI, 60, 69, 93, 127, 135, 191
    applikasjoner, 118, 130, 141, 154
    beslutningspunkter, 243
    konversasjons, 29
    modell, 93, 147, 148, 150
    sammensatte systemer, 28, 31
    samtale, 6
allmenningens tragedie, 181
Alpaca, 12
Altman, Sam, 16
Amazon Web Services, 239
Anthropic, 21, 36, 68, 122, 129
antropomorfisme, 64
API-er, 116
APIer, 67
APIs, 145
application design and frameworks, 187
applikasjonsutvikling, 208
arrays, 123
asynkron behandling, 236
Automatisk fortsettelse, 151
automatisk skalering, 238
autoregressiv modellering, 40

beredskapsplanlegging, 30
BERT, 12, 22
beslutning
    -taking scenarioer, 125
    punkter, 232
    trær, 209
beslutningsevner, 93
boundary conditions, 241
briller for utvidet virkelighet, 206
Brotli, 239, 240
Brukergrensesnitt
    grensesnitt, 201
    rammeverk, 202
Brukergrensesnitt (UI)
    design, 206
    teknologier, 197
brukeropplevelse, 184
brukerpsykologi, 203
brukertesting og tilbakemelding, 186
brukertillit, 204
brukervennlighetsproblemer, 203
Byte Pair Encoding (BPE), 12, 13

C (Programmeringsspråk), 110

Capybara-biblioteket, 245
Chain of Thought (CoT), 131
chaining of AI workers, 105
chatbot-applikasjon, 112
ChatGPT, 28, 49
Claude, 7, 40, 72
Claude 3, 46, 119, 122, 127, 129
Claude 3 Opus, 69
Claude v1, 15
Claude v2, 15
Cohere (LLM-leverandør), 21, 23
conceptual and practical challenges, 188
concurrent workflows, 240
content
    Content Categorization, 105
context
    Contextual Content Generation, 188, 189
    Contextual Field Suggestions, 189

data
    analyse, 31, 139
    behandlingsoppgaver, 118
    behandlingspipeline, 227
    Datahenting, 103
    Datasynkronisering, 103
    Datavalidering, 245
    flow, 103
    integritet, 227
    klargjøring, 102
    persistens, 103
    personvern, 24, 203
databaser, 116
    -støttet objekt, 99
databases
    locking strategies, 103
Databricks-ansatte, 48
Datadog, 235
destilleringsprosess, 71
detaljert logging, 234
deterministisk oppførsel, 54
digitalt landskap, 183
distribuert arkitektur, 236
Dohan, et al., 40
dokumentklynging, 113
Dynamisk oppgaveruting, 211
dynamisk UI-generering, 178
Dynamisk Verktøyvalg, 123

e-handel, 181, 209
E-handelapplikasjoner, 86
effektivitet, 210
eksperimentering
    rammeverk, 183
eksterne tjenester eller API-er, 119
ELK stack, 104
emosjonell tone, 137
ende-til-ende-testing, 245, 246
ensembler, 110, 111
    ensemble av arbeidere, 111
enterpriseapplikasjonsarkitektur, 35
errors
    handling, 103, 241
    Intelligent feilhåndtering, 135

rates, 104
ethics
implications, 188

F#, 87
Facebook, 22
fallback strategies, 103
feil
gjenoppretting, 245
håndtering, 101, 134
feilsøking, 212
og problemløsing, 234
og testing, 124
few-shot
læring, 58
prompting, 59
finaliseringsmetode, 147
finalize-metode, 150
finjustering, 75
FitAI, 199
flaskehalser, 213
fleksibilitet og kreativitet, 185
flertallsavstemning, 110
Flerarent
Problemløsere, 29
forklarbarhet, 244
forretningsregler, 209
Forsikringsverifisering, 96
forsyningskjede
optimalisering, 30
funksjon
anrop, 149
anropshistorikk, 148
kalling, 116
navn, 146
funksjonell programmering, 86
funksjonsanrop
feil, 126

Gemma 7B, 10
Generativ UI (GenUI), 194, 197, 201
Generative Pre-trained Transformer (GPT), 7, 63
Generative UI (GenUI), 187, 205
GitLab, 87
Gjenfinnings-forsterket generering (RAG), 43
gjenfinningsbaserte modeller, 6
Gjenfinningsforsterket Generering (RAG), 29
Gjenfinningsforsterket generering (RAG), 74
gjennomstrømning, 25
Global Interpreter Lock (GIL), 108
Google, 21
API, 59, 61
Cloud AI Platform, 22
Cloud Platform, 239
Gemini, 20
Gemini 1.5 Pro, 12, 16, 17
PaLM (Pathways Language Model), 16, 22
T5, 12
GPT-3, 12, 15

GPT-4, 6, 12, 15, 16, 19, 28, 40, 46, 58, 98, 110, 113, 120, 126, 192, 193, 237
grafiske modeller, 40
Graham, Paul, 17
grammatiske regler, 4
GraphQL, 101
Groq, 24, 113
grunnmodeller, 50
gzip, 240

hash, 144
hendelsesdrevet arkitektur, 102
henvendelsestildeling, 227
historiske mønstre, 212
Hohpe, Gregor, 98
Honeybadger, 88
HTTP, 142
hurtigbufring, 237
hyperparameter, 43
høyhastighets-fullføring, 24

ikke-overvåket læring, 4
inclusive interfaces, 188
Inferens, 5
informasjon
    gjenfinning, 6, 118
    uthenting, 49
informatikk, 65, 68
inndata
    prompts, 52
inngangsparametere, 121
innhold
    filtrering, 24
innholdsbasert filtrering, 86
Innsamling av medisinsk historie, 95
innsnevre stien, 35, 36
input
    validation, 240
instruksjonsfinjustering, 9
instruksjonsjustering
    instruksjonsjusterte modeller, 46
instruksjonstilpasning
    instruksjonstunede modeller, 48
Integrasjonsmønstre for Virksomheter, 98
integrasjonstesting, 242
integrering av LLM, 178
intelligent arbeidsflytorkestrering, 208, 216, 237
Intelligent innholdsmoderering, 220
intelligent workflow orchestration, 240
internasjonalisering, 184
iterativ forbedring, 71, 136

JSON (JavaScript Object Notation), 119, 123, 124, 127, 139, 157

K-means, 115
kanttilfeller, 54
KI, 121, 142, 198
    konversasjon, 199
    modell, 84, 198
klassifisering, 49, 113
Klinisk beslutningsstøtte, 97
kollaborativ filtrering, 86
kommandolinje
    Kommandolinjegrensesnitt (CLI), 23

komplekse oppgaver, 138
konsistens
og reproduserbarhet, 125
kontekst
Forsterkning, 43
kontekstuell beslutningstaking, 212
Kontekstuell innholdsgenerering, 177, 181–183
uendelig lange inputs, 14
vindu, 14, 212
Kontinuerlig integrasjon og distribusjon (CI/CD), 246
pipeline, 246
Kontinuerlig risikoovervåking, 97
konto, 85
kreativ skriving, 31, 49
kretsbryterlogikk, 153
kryssmodal generering, 20
Kundesentimentanalyse, 94
kundeservice-chatboter, 31
kundesupport, 29
kunnskapsbaser, 7
kunnskapshåndtering, 29
Kvantisering, 26
kvikksølv (grunnstoff), 41

language
Language Detection, 105
models, 61
Large Language Model (LLM), 16, 104, 136, 187
latens, 25
Latent Dirichlet-allokering, 115
latent rom, 37, 39
leverandører av åpen kildekode-modellverting, 193
lineær algebra, 40
lineær regresjon, 40
Llama, 12
Llama 2-70B, 46
Llama 3 70B, 10
Llama 3 8B, 10
loggoppbevaring og -rotasjon, 235
lokale utviklingsmiljøer, 146
Louvre, 39
lukket og åpen spørsmålsbesvarelse, 49

Managed Streaming for Apache Kafka, 38
Mangfold av arbeidere, 112, 157
manuell intervensjon, 215
Markdown, 139
markupbasert tagging, 66
maskinvare, 26
medisinske oppdagelser, 95
Memorial Sloan Kettering Cancer Center, 38
Menneske-i-løkken (HITL), 169
Merkur (planet), 41
Merkur (romersk gud), 41
MessagePack, 239
Meta, 22
Metropolitan Museum of Art, 39
Mikrotjeneste-arkitektur, 84
Mistral, 23

7B, 10
7B Instruct, 15, 193
Mixtral
8x22B, 10
8x7B, 52
moderne applikasjoner, 210
modularitet, 83
monitoring
and logging, 104
motivasjonsstrategier, 201
multi-step workflow, 105
Multimodal
modeller, 18
språkmodeller, 19
mønstergjenkjenning, 144

Naiv Bayes, 114
narrativ konstruksjon, 18
naturlig språk
Naturlig språkprosessering (NLP), 95, 113
nettbrett, 206
nettbutikker, 193
nettverkstilkobling, 213
nevrale nettverk, 3, 6
New Relic, 238
nullskudds-læring, 54
nøkkelmønstre, 211

Ollama, 23
Olympia, 31, 58, 121, 135, 143, 158
Olympias kunnskapsbase, 86
One-Shot Learning, 56
OpenAI, 3, 21, 36, 68
OpenRouter, 25, 26, 143, 237
oppsummering, 49
OPT model, 22
optimistic locking, 103
ordbøker, 123
output verification, 241
oversettelse, 15, 185
overvåking
metrikker, 234
og logging, 233
og varsling, 214

parafrasering, 49
parallell utførelse, 236
parameter
effekter, 121
område, 10
Parameterantall, 26
Perplexity (Leverandør), 10
personalisering, 178, 205, 210
Personalisert mikrotekst, 194
personaliserte produktanbefalinger, 86
personalization
Personalized Forms, 189
pessimistic locking, 103
prediksjoner, 5
prinsippet om minste privilegium, 67
probabilistiske modeller, 40
Process Manager
Enterprise Integration, 216
processing time, 104

Produktanbefalinger, 86
Produktivitet, 180
programvarearkitektur, 2
progressiv avsløring, 195
prompts
    design, 54
    engineering, 55, 61
    forbedring, 64
    kjeding, 55, 67
    konstruksjon, 37, 42, 62
    Prompt Template, 193
    Prompt-destillering, 43, 73, 237
    Prompt-mal, 55
    Promptdestillering, 68
    Promptobjekt, 69
    teknikk, 202
    utforming, 52, 63
Prosesshåndterer, 98, 101
Protocol Buffers, 239
publiser-abonner-systemer, 102
PyTorch, 22

Qwen2 70B, 10

Rails, 184
Railway Oriented Programming (ROP), 89
Raix, 217
    bibliotek, 91
rangerere, 33
repetisjonsstraffer, 48
Responsavgrensning, 167
Response Fencing, 193
Result Interpreter, 134
Retrieval Augmented Generation (RAG), 35, 118
retry mechanisms, 103
revisjon og etterlevelse, 234
revisjonslogging, 100
risikofaktorer, 90, 91
Risikostratifisering, 96
rollespill-lignende interaksjoner, 6
RSpec, 241, 242, 245
Ruby, 87, 88, 106, 154, 245
Ruby on Rails, 1, 105, 216, 224
Rudall, Alex, 21
Rust (Programmeringsspråk), 110
Rust (Programming Language), 87

samtale
    logg, 148, 151
    løkke, 149, 151
satsvis behandling, 237
Scout, 238
segmenterings- og målrettingsstrategier, 183
Selvhelbredende Data, 230
Selvhelbredende data, 155
sentiment analysis, 105
sentimentanalyse, 94, 108, 127
serversendte hendelser (SSE), 142
sinnsteori, 37
skalerbarhet, 210, 235
skjevhet
    og rettferdighet i AI, 244
smarttelefoner, 206

sporing av viktige målinger, 231
språk
-relaterte oppgaver, 4
modeller, 39, 68
spørsmål-og-svar-systemer, 7
SQL-injeksjoner, 66
stasjonære datamaskiner, 206
stemmestyrte grensesnitt, 31
stemningsanalyse, 15, 106, 111, 137
Stor språkmodell (LLM), 1, 3, 62, 64, 66, 67, 71, 117, 132, 192
Store språkmodeller (LLM), 14, 27, 113, 177, 197
landskap, 25
Stort språkmodell (LLM), 72, 82, 116, 126, 136, 138, 155, 158, 219
Stripe, 122
Structured IO, 193
strukturert logging, 235
strukturerte data, 126
strømbehandling, 154
strømhåndterere, 142
strømmebehandling, 142
logikk, 150
strømmedata, 144
strømprosessering, 147
Støttevektormaskiner (SVM), 114
svindeldeteksjon
system, 91
Symptomvurdering og stratifisering, 95
syntaksfeil, 124
syntetisk datagenerering, 49
systemdirektiv, 93, 121

T5, 22
Tankerekke (CoT), 42
temaidentifisering, 113
Temperatur, 50
testmiljøer, 247
Text Cleanup, 105
Tid til første token (TTFT), 25
tilbakemelding
Tilbakemeldingssløyfe, 55
tilbakerullingsmekanismer, 247
tilgjengelighet, 204, 205
tilpasning, 25
tilstandsløs, 148
Tilstedeværelsesstraff, 45
Together.ai, 24
tokener, 11
tokenisering, 11
tokens, 5
Top-k sampling, 45
Top-p (nucleus) sampling, 45
trafikkstyring, 30
transformerarkitektur, 6
treningsdata, 39
Tvunget Verktøyvalg, 124

Unicode-kodbare språk, 13
Universal ID, 239
unntakshåndtering, 213, 215
User Interface (UI)
interfaces, 187
user-generated content, 105

utdanningsapplikasjoner, 29
utløsermelding, 98
utviklingsrammeverk, 140

Ventriloquist, 167
verktøybruk, 116, 140
verktøykall, 145
virtuelle assistenter, 31
visuelt grensesnitt, 197

Wall, Larry, 3
Wisper, 88, 100, 143, 150
Wooley, Chad, 87

XML, 126

Yi-34B, 46
ytelse
    kompromisser, 5
    optimalisering, 125, 185, 234
    problemer, 238

zero-shot learning, 55

økosystem, 139