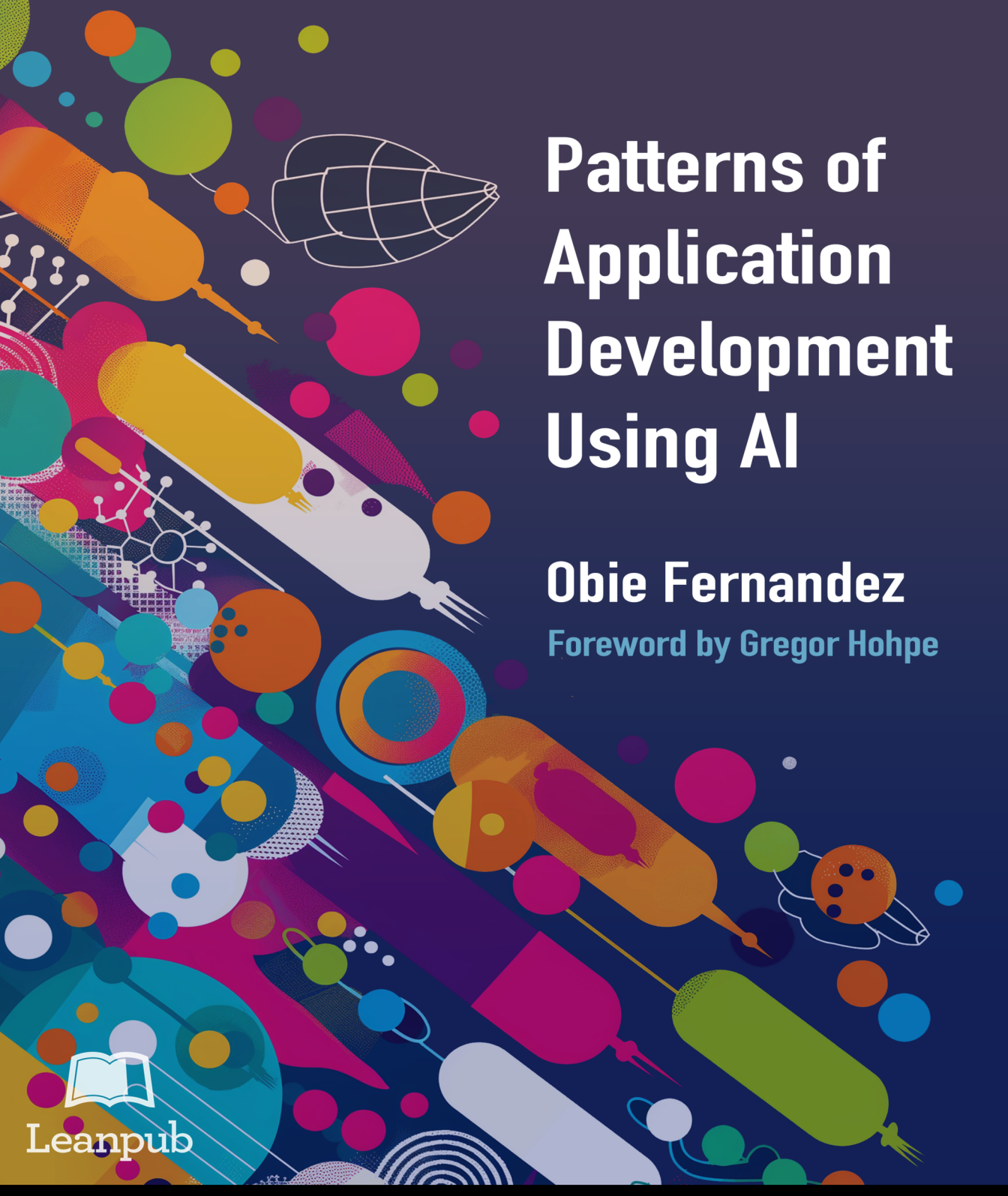



**Dansk Udgave**

# Mønstre i Applikationsudvikling med AI (Dansk Udgave)

Obie Fernandez

Denne bog er til salg på
http://leanpub.com/patterns-of-application-development-using-ai-da

Denne version blev offentliggjort den 2025-01-23


Dette er en Leanpub bog. Leanpub giver forfattere og udgivere magten med Lean Publishing-processen. Lean Publishing er handlingen med at publicere en igangværende e-bog ved hjælp af letvægtsværktøjer og mange iterationer for at få læserfeedback, pivotere indtil du har den rigtige bog og opbygge trækkraft, når du gør det.

# Tweet Denne Bog!

Hjælp venligst Obie Fernandez med at sprede ordet om denne bog på Twitter!

Den foreslåede hashtag for denne bog er #poaduai.

Find ud af, hvad andre mennesker siger om bogen ved at klikke på dette link for at søge efter denne hashtag på Twitter:

#poaduai

*Til min seje dronning, min muse, mit lys og min kærlighed, Victoria*

# Også Af 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

# Indhold

# Forord af Gregor Hohpe

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Forord

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Om Bogen

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Om Kodeeksemplerne

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvad Jeg Ikke Dækker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvem Denne Bog Er Til

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Opbygning af et Fælles Ordforråd

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Bliv Involveret

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Tak

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvad med illustrationerne?

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

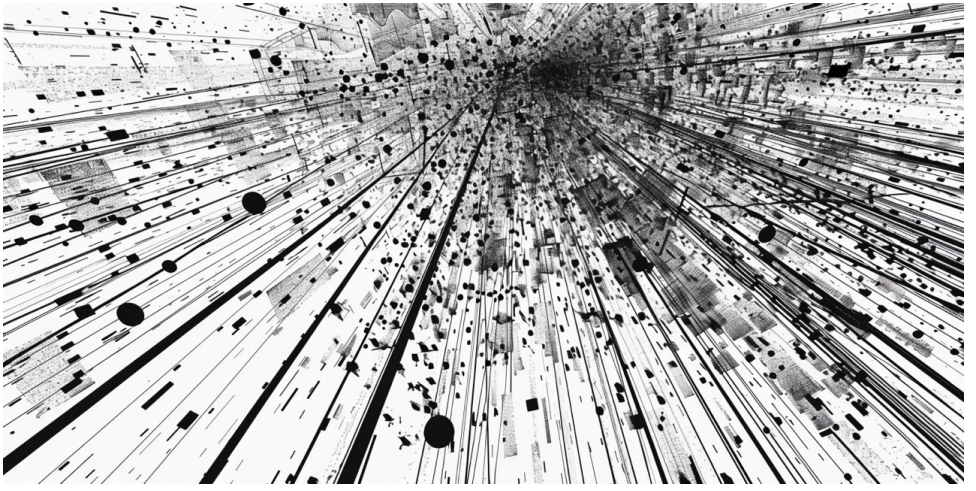
## Om Lean Publishing

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Om forfatteren

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Introduktion



Hvis du er ivrig efter at begynde at integrere AI Store Sprogmodeller (LLMs) i dine programmeringsprojekter, er du velkommen til at springe direkte til mønstrene og kodeeksemplerne i de senere kapitler. For at få fuldt udbytte af disse mønstre og deres potentiale er det dog værd at bruge et øjeblik på at forstå den bredere kontekst og den sammenhængende tilgang, de repræsenterer.

Mønstrene er ikke blot en samling af isolerede teknikker, men snarere et samlet rammeværk for integration af AI i dine applikationer. Jeg bruger Ruby on Rails, men disse mønstre burde virke i stort set ethvert andet programmeringsmiljø. De adresserer en bred vifte af områder, fra datahåndtering og ydelsesforbedring til brugeroplevelse og sikkerhed, og giver dermed et omfattende sæt værktøjer til at forbedre traditionel programmeringspraksis med AI's muligheder.

Hver kategori af mønstre tackler en specifik udfordring eller mulighed, der opstår, når man indbygger AI-komponenter i sin applikation. Ved at forstå relationerne og

synergierne mellem disse mønstre kan du træffe informerede beslutninger om, hvor og hvordan AI kan anvendes mest effektivt.

Mønstre er aldrig præskriptive løsninger og bør ikke behandles som sådan. De er tænkt som tilpasningsdygtige byggeklodser, der skal skræddersys til de unikke krav og begrænsninger i din egen applikation. Den vellykkede anvendelse af disse mønstre (ligesom alle andre inden for softwareområdet) afhænger af en dyb forståelse af problemdomænet, brugernes behov og den overordnede tekniske arkitektur i dit projekt.

## Tanker om Softwarearkitektur

Jeg begyndte at programmere i 1980'erne og var involveret i hackermiljøet, og jeg har aldrig mistet min hackermentalitet, selv efter jeg blev professionel softwareudvikler. Lige fra starten har jeg altid haft en sund skepsis over for, hvilken værdi softwarearkitekter i deres elfenbenstårne faktisk bragte til bordet.

En af grundene til, at jeg personligt er så begejstret for de forandringer, som denne kraftfulde nye bølge af AI-teknologi medfører, er dens indvirkning på det, vi betragter som *softwarearkitektur*-beslutninger. Den udfordrer traditionelle opfattelser af, hvad der udgør den "korrekte" måde at designe og implementere vores softwareprojekter på. Den sætter også spørgsmålstegn ved, om arkitektur stadig primært kan betragtes som *de dele af et system, der er svære at ændre*, eftersom AI-forbedringer gør det nemmere end nogensinde at ændre enhver del af dit projekt når som helst.

Måske er vi på vej ind i højdepunktet af den "postmoderne" tilgang til softwareudvikling. I denne sammenhæng henviser postmoderne til et fundamentalt skift væk fra traditionelle paradigmer, hvor udviklere var ansvarlige for at skrive og vedligeholde hver eneste kodelinje. I stedet omfavner den idéen om at delegere opgaver som datamanipulation, komplekse algoritmer og endda hele dele af applikationslogikken til tredjepartsbiblioteker og eksterne API'er. Dette postmoderne skift repræsenterer en betydelig afvigelse fra den konventionelle visdom om at bygge applikationer fra bunden, og det udfordrer udviklere til at gentænke deres rolle i udviklingsprocessen.

Jeg har altid ment, at gode programmører kun skriver den kode, der er absolut nødvendig at skrive, baseret på læren fra Larry Wall og andre hackerkoryfæer som ham. Ved at minimere mængden af skrevet kode kan vi bevæge os hurtigere, reducere overfladearealet for fejl, forenkle vedligeholdelsen og forbedre den generelle pålidelighed af vores applikationer. Mindre kode giver os mulighed for at fokusere på kerneforretningslogikken og brugeroplevelsen, mens andet arbejde delegeres til andre tjenester.

Nu hvor AI-drevne systemer kan håndtere opgaver, der tidligere var forbeholdt menneskeskrevet kode, burde vi kunne være endnu mere produktive og agile, med større fokus end nogensinde på at skabe forretningsværdi og brugeroplevelse.

Naturligvis er der kompromiser ved at delegere store dele af dit projekt til AI-systemer, såsom potentielt tab af kontrol og behovet for robust overvågning og feedback-mekanismer. Det er derfor, det kræver et nyt sæt færdigheder og viden, herunder i det mindste en grundlæggende forståelse af, hvordan AI fungerer.

## Hvad er en Store Sprogmodel?

Store Sprogmodeller (LLMs) er en type kunstig intelligens-model, der har fået betydelig opmærksomhed i de senere år, siden lanceringen af GPT-3 af OpenAI i 2020. LLMs er designet til at behandle, forstå og generere menneskeligt sprog med bemærkelsesværdig præcision og flydende. I dette afsnit vil vi kort se på, hvordan LLMs fungerer, og hvorfor de er velegnede til at bygge intelligente systemkomponenter.

I deres kerne er LLMs baseret på algoritmer inden for dyb læring, specifikt neurale netværk. Disse netværk består af sammenkoblede knudepunkter eller neuroner, der behandler og overfører information. Den foretrukne arkitektur for LLMs er ofte Transformer-modellen, som har vist sig at være meget effektiv til at håndtere sekventielle data som tekst.

Transformer-modeller er baseret på opmærksomhedsmekanismen og bruges primært

til opgaver med sekventielle data, såsom naturlig sprogbehandling. Transformers behandler inputdata på én gang i stedet for sekventielt, hvilket gør dem i stand til at opfange langdistanceafhængigheder mere effektivt. De har lag af opmærksomhedsmekanismer, der hjælper modellen med at fokusere på forskellige dele af inputdataene for at forstå kontekst og relationer.

Træningsprocessen for store sprogmodeller involverer at eksponere modellen for enorme mængder tekstdata, såsom bøger, artikler, hjemmesider og kodearkiver. Under træningen lærer modellen at genkende mønstre, relationer og strukturer i teksten. Den opfanger sprogets statistiske egenskaber, såsom grammatiske regler, ordassociationer og kontekstuelle betydninger.

En af de vigtigste teknikker, der bruges i træningen af store sprogmodeller, er ikke-superviseret læring. Dette betyder, at modellen lærer fra data uden eksplicit mærkning eller vejledning. Den opdager mønstre og repræsentationer på egen hånd ved at analysere samforekomsten af ord og fraser i træningsdataene. Dette giver store sprogmodeller mulighed for at udvikle en dyb forståelse af sprog og dets nuancer.

Et andet vigtigt aspekt ved store sprogmodeller er deres evne til at håndtere *kontekst*. Når de behandler et stykke tekst, tager store sprogmodeller ikke kun de enkelte ord i betragtning, men også den omgivende kontekst. De tager højde for de foregående ord, sætninger og endda afsnit for at forstå tekstens betydning og hensigt. Denne kontekstuelle forståelse gør store sprogmodeller i stand til at generere sammenhængende og relevante svar. En af de primære måder, hvorpå vi evaluerer en given sprogmodels kapacitet, er ved at overveje størrelsen af den kontekst, de kan tage i betragtning for at generere svar.

Når de er trænet, kan store sprogmodeller bruges til en lang række sprogrelaterede opgaver. De kan generere menneskelig tekst, besvare spørgsmål, opsummere dokumenter, oversætte sprog og endda skrive kode. Store sprogmodellers alsidighed gør dem værdifulde til at opbygge intelligente systemkomponenter, der kan interagere med brugere, behandle og analysere tekstdata og generere meningsfuldt output.

Ved at inkorporere store sprogmodeller i applikationsarkitekturen kan du skabe AI-komponenter, der forstår og behandler brugerinput, genererer dynamisk indhold og leverer intelligente anbefalinger eller handlinger. Men at arbejde med store sprogmodeller kræver omhyggelig overvejelse af ressourcekrav og ydelses-kompromiser. Store sprogmodeller er beregningstunge og kan kræve betydelig processorkraft og hukommelse (med andre ord, penge) at drive. De fleste af os bliver nødt til at vurdere omkostningsimplikationerne ved at integrere store sprogmodeller i vores applikationer og handle derefter.

## Forståelse af inferens

Inferens refererer til den proces, hvorved en model genererer forudsigelser eller output baseret på nye, usete data. Det er den fase, hvor den trænede model bruges til at træffe beslutninger eller generere tekst, billeder eller andet indhold som svar på brugerinput.

Under træningsfasen lærer en AI-model fra et stort datasæt ved at justere sine parametre for at minimere fejlen i sine forudsigelser. Når modellen er trænet, kan den anvende det, den har lært, på nye data. Inferens er hvordan modellen bruger sine lærte mønstre og viden til at generere output.

For store sprogmodeller involverer inferens at tage et prompt eller inputtekst og producere et sammenhængende og kontekstuelt relevant svar, som en strøm af *tokens* (som vi snart vil tale om). Dette kunne være at besvare et spørgsmål, fuldføre en sætning, generere en historie eller oversætte tekst, blandt mange andre opgaver.

I modsætning til den måde, du og jeg tænker på, sker en AI-models "tænkning" via inferens i én samlet tilstandsløs operation. Det vil sige, at dens tænkning er begrænset til dens genereringsproces. Den er bogstaveligt talt nødt til at tænke højt, som hvis jeg stillede dig et spørgsmål og kun accepterede et svar fra dig i "stream of consciousness"-stil.

## Store sprogmodeller kommer i mange størrelser og varianter

Mens praktisk talt alle populære store sprogmodeller er baseret på den samme grundlæggende transformer-arkitektur og trænet på enorme tekstdatasæt, kommer de i forskellige størrelser og er finjusteret til forskellige formål. Størrelsen på en stor sprogmodel, målt i antallet af parametre i dens neurale netværk, har stor indflydelse på dens kapaciteter. Større modeller med flere parametre, som GPT-4, der rygtes at have 1 til 2 billioner parametre, er generelt mere vidende og kapable end mindre modeller. Dog kræver større modeller også meget mere computerkraft at køre, hvilket oversættes til højere udgifter, når du bruger dem via API-kald.

For at gøre store sprogmodeller mere praktiske og skræddersyede til specifikke anvendelser bliver basismodellerne ofte finjusteret på mere målrettede datasæt. For eksempel kan en stor sprogmodel trænes på et stort korpus af dialog for at specialisere den til konversations-AI. Andre er trænet på kode for at give dem programmeringsviden. Der er endda modeller, der er særligt trænet til rollespilslignende interaktioner med brugere!

## Genfinding vs Generative Modeller

I verden af store sprogmodeller (LLMs) findes der to hovedtilgange til at generere svar: genfindingsbaserede modeller og generative modeller. Hver tilgang har sine egne styrker og svagheder, og forståelsen af forskellene mellem dem kan hjælpe dig med at vælge den rigtige model til dit specifikke anvendelsesformål.

### Genfindingsbaserede Modeller

Genfindingsbaserede modeller, også kendt som informationsgenfindingsmodeller, genererer svar ved at søge gennem en stor database af eksisterende tekst og udvælge de mest relevante passager baseret på input-forespørgslen. Disse modeller genererer ikke

ny tekst fra bunden, men sammensætter i stedet uddrag fra databasen for at danne et sammenhængende svar.

En af de største fordele ved genfindingsbaserede modeller er deres evne til at levere faktuelt korrekt og opdateret information. Da de er afhængige af en database med kurateret tekst, kan de hente relevant information fra pålidelige kilder og præsentere den for brugeren. Dette gør dem velegnede til applikationer, der kræver præcise, faktuelle svar, såsom spørgsmål-svar-systemer eller videnbaser.

Genfindingsbaserede modeller har dog nogle begrænsninger. De er kun så gode som den database, de søger i, så databasens kvalitet og dækning påvirker direkte modellens ydeevne. Derudover kan disse modeller have svært ved at generere sammenhængende og naturligt lydende svar, da de er begrænset til den tekst, der er tilgængelig i databasen.

Vi dækker ikke brugen af rene genfindingsmodeller i denne bog.

## Generative Modeller

Generative modeller skaber derimod ny tekst fra bunden baseret på de mønstre og sammenhænge, de har lært under træningen. Disse modeller bruger deres forståelse af sprog til at generere nye svar, der er skræddersyet til input-prompten.

Den største styrke ved generative modeller er deres evne til at producere kreativ, sammenhængende og kontekstuelt relevant tekst. De kan deltage i åbne samtaler, generere historier og endda skrive kode. Dette gør dem ideelle til applikationer, der kræver mere åbne og dynamiske interaktioner, såsom chatbots, indholdsproduktion og kreative skriveassistenter.

Generative modeller kan dog nogle gange producere inkonsistent eller faktuelt ukorrekt information, da de er afhængige af de mønstre, der er lært under træningen, frem for en kurateret database med fakta. De kan også være mere tilbøjelige til bias og hallucinationer, hvor de genererer tekst, der er plausibel, men ikke nødvendigvis sand.

Eksempler på generative LLMs inkluderer OpenAI's GPT-serie (GPT-3, GPT-4) og Anthropics Claude.

## Hybridmodeller

Flere kommercielt tilgængelige LLMs kombinerer både genfinding og generative tilgange i en hybridmodel. Disse modeller bruger genfindingsteknikker til at finde relevant information fra en database og bruger derefter generative teknikker til at sammenfatte denne information til et sammenhængende svar.

Hybridmodeller sigter mod at kombinere den faktuelle nøjagtighed fra genfindingsbaserede modeller med de naturlige sproggenereringsevner fra generative modeller. De kan levere mere pålidelig og opdateret information, mens de stadig bevarer evnen til at deltage i åbne samtaler.

Når du vælger mellem genfindingsbaserede og generative modeller, bør du overveje de specifikke krav til din applikation. Hvis det primære mål er at levere præcis, faktuel information, kan en genfindingsbaseret model være det bedste valg. Hvis applikationen kræver mere åbne og kreative interaktioner, kan en generativ model være mere passende. Hybridmodeller tilbyder en balance mellem de to tilgange og kan være et godt valg til applikationer, der kræver både faktuel nøjagtighed og naturlig sproggenerering.

I sidste ende afhænger valget mellem genfindingsbaserede og generative modeller af den specifikke anvendelse og afvejningen mellem nøjagtighed, kreativitet og fleksibilitet. Ved at forstå styrkerne og begrænsningerne ved hver tilgang kan du træffe informerede beslutninger.

## Instruktionstilpasning

Instruktionstilpasning er en undergruppe af samtaletræning, der bruger menneskeforfattede prompts og svar til at gøre LLMs bedre til at forstå og følge instruktioner. Her er nogle simple eksempler, der illustrerer, hvordan instruktionstilpasning på menneskeforfattede prompts og svar gør LLMs bedre til at forstå og følge instruktioner:

### Eksempel 1: Kundeservice

Instruktion: "Giv et svar til en kunde, der beder om refundering."

Menneskeforfattet Prompt: "Jeg vil gerne anmode om en refundering for mit nylige køb. Produktet levede ikke op til mine forventninger."

Menneskeforfattet Svar: "Jeg beklager at høre, at produktet ikke levede op til dine forventninger. For at behandle din refundering bedes du oplyse dit ordrenummer og årsagen til returneringen. Vi vil gennemgå din anmodning og vende tilbage til dig inden for 2-3 arbejdsdage."

### Eksempel 2: Uddannelsesmæssig Assistance

Instruktion: "Forklar konceptet fotosyntese til en 10-årig."

Human-Written Prompt: "Kan du forklare, hvordan planter laver deres mad?"

Human-Written Response: "Selvfølgelig! Planter laver deres mad gennem en proces, der kaldes fotosyntese. De bruger sollys, vand og kuldioxid til at skabe mad og ilt. Plantens blade har en særlig del, der kaldes klorofyl, som hjælper dem med at indfange sollys."

### Example 3: Personlig Assistent

> Instruction: "Planlæg et møde til næste tirsdag klokken 15."
>
> Human-Written Prompt: "Kan du arrangere et møde til tirsdag klokken 15?"
>
> Human-Written Response: "Selvfølgelig, jeg har planlagt et møde til dig næste tirsdag klokken 15. Er der andet, du har brug for?"

Resultatet er et mangfoldigt økosystem af LLM'er i forskellige størrelser og med forskellige specialer. Mindre modeller i området 1-7 milliarder parametre giver gode generelle sproglige evner, samtidig med at de er mere effektive at køre.

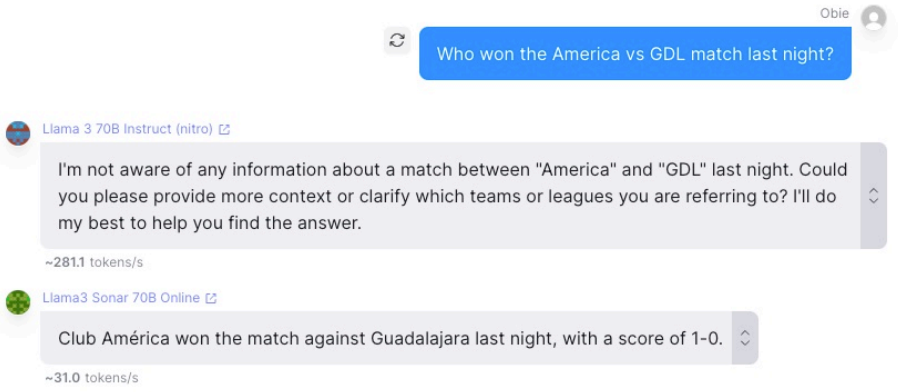
- Mistral 7B
- Llama 3 8B
- Gemma 7B

Mellemstore modeller omkring 30-70 milliarder parametre tilbyder stærkere ræsonnements- og instruktionsfølgende evner.

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

Når man vælger en LLM til at indbygge i en applikation, skal man afbalancere modellens kapaciteter mod praktiske faktorer som omkostninger, latenstid, kontekstlængde og indholdsfiltrering. Mindre, instruktionstilpassede modeller er ofte det bedste valg til enklere sprogopgaver, mens de største modeller kan være nødvendige til kompleks ræsonnement eller analyse. Modellens træningsdata er også en vigtig overvejelse, da det bestemmer modellens vidensskæringsdato.

Visse modeller, som nogle fra Perplexity, er forbundet til realtids informationskilder, så de reelt set ikke har nogen skæringsdato. Når du stiller dem spørgsmål, kan de selvstændigt beslutte at foretage websøgninger og hente vilkårlige websider for at generere et svar.

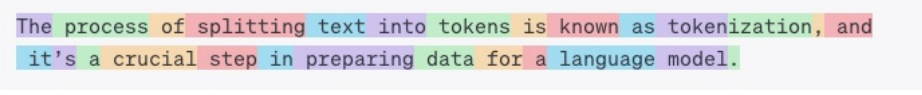



**Figur 1. Llama3 med og uden online adgang**

I sidste ende findes der ikke én LLM, der passer til alle formål. At forstå variationerne i modelstørrelse, arkitektur og træning er nøglen til at vælge den rigtige model til en given anvendelse. At eksperimentere med forskellige modeller er den eneste praktiske måde at afdække, hvilke der giver den bedste ydeevne til den pågældende opgave.

## Tokenisering: At opdele tekst i stykker

Før en stor sprogmodel kan behandle tekst, skal teksten opdeles i mindre enheder kaldet *tokens*. Tokens kan være enkelte ord, dele af ord eller endda enkelte tegn. Processen med at opdele tekst i tokens kaldes tokenisering, og det er et afgørende trin i forberedelsen af data til en sprogmodel.




**Figur 2. Denne sætning indeholder 27 tokens**

Forskellige LLM'er bruger forskellige tokeniseringsstrategier, hvilket kan have betydelig indflydelse på modellens ydeevne og kapaciteter. Nogle almindelige tokenizers, der

bruges af LLM'er, omfatter:

- **GPT (Byte Pair Encoding)**: GPT-tokenizers bruger en teknik kaldet byte pair encoding (BPE) til at opdele tekst i delord-enheder. BPE sammenlægger iterativt de hyppigst forekommende byte-par i et tekstkorpus og danner derved et ordforråd af delord-tokens. Dette gør det muligt for tokenizeren at håndtere sjældne og nye ord ved at opdele dem i mere almindelige delord-stykker. GPT-tokenizers bruges af modeller som GPT-3 og GPT-4.

- **Llama (SentencePiece)**: Llama-tokenizere bruger SentencePiece-biblioteket, som er en ikke-superviseret teksttokenizer og detokenizer. SentencePiece behandler inputteksten som en sekvens af Unicode-tegn og lærer et delordsvokabular baseret på et træningskorpus. Det kan håndtere ethvert sprog, der kan kodes i Unicode, hvilket gør det velegnet til flersprogede modeller. Llama-tokenizere bruges af modeller som Metas Llama og Alpaca.

- **SentencePiece (Unigram)**: SentencePiece-tokenizere kan også bruge en anden algoritme kaldet Unigram, som er baseret på en delords-regulariseringsteknik. Unigram-tokenisering bestemmer det optimale delordsvokabular baseret på en unigram-sprogmodel, som tildeler sandsynligheder til individuelle delordsenheder. Denne tilgang kan producere mere semantisk meningsfulde delord sammenlignet med BPE. SentencePiece med Unigram bruges af modeller som Googles T5 og BERT.

- **Google Gemini (Multimodal Tokenisering)**: Google Gemini bruger et tokeniseringsskema designet til at håndtere forskellige datatyper, herunder tekst, billeder, lyd, videoer og kode. Denne multimodale kapacitet gør det muligt for Gemini at behandle og integrere forskellige former for information. Særligt bemærkelsesværdigt har Google Gemini 1.5 Pro et kontekstvindue, der kan håndtere millioner af tokens, meget større end tidligere modeller. Dette

omfattende kontekstvindue gør det muligt for modellen at behandle en større kontekst, hvilket potentielt fører til mere præcise svar. Det er dog vigtigt at bemærke, at Geminis tokeniseringsskema er meget tættere på ét token pr. tegn end andre modeller. Dette betyder, at de faktiske omkostninger ved at bruge Gemini-modeller kan være betydeligt højere end forventet, hvis du er vant til at bruge modeller som GPT, da Googles prissætning er baseret på tegn frem for tokens.

Valget af tokenizer påvirker flere aspekter af en LLM, herunder:

- **Vokabularstørrelse**: Tokenizeren bestemmer størrelsen af modellens vokabular, som er sættet af unikke tokens, den genkender. Et større, mere detaljeret vokabular kan hjælpe modellen med at håndtere en bredere vifte af ord og fraser og endda blive multimodal (i stand til at forstå og generere mere end bare tekst), men det øger også modellens hukommelseskrav og beregningsmæssige kompleksitet.
- **Håndtering af sjældne og ukendte ord**: Tokenizere, der bruger delordsenheder, som BPE og SentencePiece, kan nedbryde sjældne og ukendte ord i mere almindelige delordsstykker. Dette gør det muligt for modellen at lave kvalificerede gæt om betydningen af ord, den ikke har set før, baseret på de delord, de indeholder.
- **Flersproget support**: Tokenizere som SentencePiece, der kan håndtere ethvert Unicode-koderbart sprog, er velegnede til flersprogede modeller, der skal behandle tekst på flere sprog.

Når man vælger en LLM til en bestemt anvendelse, er det vigtigt at overveje, hvilken tokenizer den bruger, og hvor godt den passer til de specifikke sprogbehandlingsbehov for den pågældende opgave. Tokenizeren kan have en betydelig indvirkning på modellens evne til at håndtere domænespecifik terminologi, sjældne ord og flersproget tekst.

# Kontekststørrelse: Hvor Meget Information Kan en Sprogmodel Bruge Under Inferens?

Når man diskuterer sprogmodeller, refererer kontekststørrelse til mængden af tekst, som en model kan overveje, når den behandler eller genererer sine svar. Det er grundlæggende et mål for, hvor meget information modellen kan "huske" og bruge til at informere sine outputs (udtrykt i tokens). Kontekststørrelsen af en sprogmodel kan have en betydelig indvirkning på dens kapaciteter og de typer opgaver, den effektivt kan udføre.

## Hvad er Kontekststørrelse?

I tekniske termer bestemmes kontekststørrelsen af antallet af tokens (ord eller orddele), som en sprogmodel kan behandle i en enkelt inputsekvens. Dette omtales ofte som modellens "opmærksomhedsspænd" eller "kontekstvindue". Jo større kontekststørrelsen er, jo mere tekst kan modellen overveje på én gang, når den genererer et svar eller udfører en opgave.

Forskellige sprogmodeller har varierende kontekststørrelser, der spænder fra nogle få hundrede tokens til millioner af tokens. Til reference kan et typisk tekstafsnit indeholde omkring 100-150 tokens, mens en hel bog kan indeholde titusinder eller hundredtusinder af tokens.

> Der er endda arbejde med effektive metoder til at skalere Transformer-baserede Store Sprogmodeller (LLM) til uendeligt lange inputs med begrænset hukommelse og beregning.

## Hvorfor er kontekststørrelse vigtig?

Kontekststørrelsen i en sprogmodel har en betydelig indflydelse på dens evne til at forstå og generere sammenhængende, kontekstuelt relevant tekst. Her er nogle vigtige grunde til, at kontekststørrelse betyder noget:

1. **Forståelse af længere indhold**: Modeller med større kontekststørrelser kan bedre forstå og analysere længere tekster, såsom artikler, rapporter eller endda hele bøger. Dette er afgørende for opgaver som dokumentsammenfatning, besvarelse af spørgsmål og indholdsanalyse.

2. **Opretholdelse af sammenhæng**: Et større kontekstvindue gør det muligt for modellen at opretholde sammenhæng og konsistens på tværs af længere output. Dette er vigtigt for opgaver som historiegenerering, dialogsystemer og indholdsproduktion, hvor det er essentielt at opretholde en konsistent fortælling eller emne. Det er også absolut afgørende, når man bruger LLM'er til at generere eller transformere strukturerede data.
3. **Opfangelse af langdistanceafhængigheder**: Nogle sprogopgaver kræver forståelse af relationer mellem ord eller sætninger, der er langt fra hinanden i en tekst. Modeller med større kontekststørrelser er bedre rustet til at opfange disse langdistanceafhængigheder, hvilket kan være vigtigt for opgaver som sentimentanalyse, oversættelse og sprogforståelse.
4. **Håndtering af komplekse instruktioner**: I anvendelser hvor sprogmodeller bruges til at følge komplekse instruktioner i flere trin, tillader en større kontekststørrelse modellen at tage hele sættet af instruktioner i betragtning, når den genererer et svar, i stedet for kun de seneste få ord.

## Eksempler på sprogmodeller med forskellige kontekststørrelser

Her er nogle eksempler på sprogmodeller med forskellige 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 der en bred vifte af kontekststørrelser blandt disse modeller, fra omkring 4.000 tokens for OpenAI GPT-3.5 Turbo-modellen til 200.000 tokens for Anthropic Claude v2-modellen. Nogle modeller, som Google's PaLM 2 og OpenAI's GPT-4, tilbyder forskellige varianter med større kontekststørrelser (f.eks. "32k"-versioner), som kan håndtere endnu længere inputsekvenser. Og i øjeblikket (april 2024) praler Google Gemini Pro med næsten 3 millioner tokens!

Det er værd at bemærke, at kontekststørrelsen kan variere afhængigt af den specifikke implementering og version af en bestemt model. For eksempel har den oprindelige OpenAI GPT-4-model en kontekststørrelse på 8.191 tokens, mens de senere GPT-4-varianter som Turbo og 4o har en meget større kontekststørrelse på 128.000 tokens.

> Sam Altman har sammenlignet nuværende kontekstbegrænsninger med de kilobyte arbejdshukommelse, som personlige computerprogrammører måtte arbejde med i 80'erne, og sagde, at vi i den nærmeste fremtid vil kunne passe "alle dine personlige data" ind i konteksten af en stor sprogmodel.

## Valg af den rigtige kontekststørrelse

Når man vælger en sprogmodel til en bestemt anvendelse, er det vigtigt at overveje opgavens kontekststørrelseskrav. For opgaver der involverer korte, isolerede

tekstdele, som sentimentanalyse eller simpel spørgsmålsbesvarelse, kan en mindre kontekststørrelse være tilstrækkelig. For opgaver der kræver forståelse og generering af længere, mere komplekse tekster, vil en større kontekststørrelse sandsynligvis være nødvendig.

Det er værd at bemærke, at større kontekststørrelser ofte medfører øgede beregningsomkostninger og langsommere behandlingstider, da modellen skal tage mere information i betragtning, når den genererer et svar. Derfor skal du finde en balance mellem kontekststørrelse og ydeevne, når du vælger en sprogmodel til din anvendelse.

> Hvorfor ikke bare vælge modellen med den største kontekststørrelse og fylde den med så meget information som muligt? Tja, ud over ydelsesfaktorer er den anden hovedovervejelse omkostningerne. I marts 2024 vil en *enkelt* prompt-respons-cyklus med Google Gemini Pro 1.5 med fuld kontekst koste dig næsten 8 dollars (USD). Hvis du har et anvendelsesformål, der retfærdiggør den udgift, så held og lykke med det! Men for de fleste anvendelser er det simpelthen for dyrt med flere størrelsesordener.

## At finde nåle i høstakke

Konceptet med at finde en nål i en høstak har længe været en metafor for udfordringerne ved udtrækning i store datasæt. Inden for store sprogmodeller justerer vi denne analogi en smule. Forestil dig, at vi ikke bare leder efter én enkelt oplysning begravet i en omfattende tekst (som en komplet antologi af Paul Graham essays), men flere oplysninger spredt ud over det hele. Dette scenarie minder mere om at finde flere nåle i en kæmpemæssig mark, ikke bare en enkelt høstak. Her kommer det interessante: vi skal ikke kun lokalisere disse nåle, men også væve dem sammen til en sammenhængende tråd.

Når store sprogmodeller får til opgave at udtrække og ræsonnere over flere oplysninger

indlejret i lange kontekster, står de over for en dobbelt udfordring. For det første er der det simple problem med præcisionen af udtrækningen - den falder naturligt, efterhånden som antallet af oplysninger stiger. Dette er forventeligt; når alt kommer til alt, belaster det selv de mest sofistikerede modeller at holde styr på flere detaljer på tværs af en omfattende tekst.

For det andet, og måske mere kritisk, er der udfordringen med at ræsonnere over disse oplysninger. Det er én ting at plukke oplysninger ud; det er noget helt andet at sammenfatte dem til et sammenhængende narrativ eller svar. Det er her, den virkelige test kommer ind. Sprogmodellernes ydeevne i ræsonnementsopgaver har en tendens til at forringes mere end i simple udtrækningsopgaver. Denne forringelse handler ikke kun om mængden; det handler om det komplekse samspil mellem kontekst, relevans og følgeslutning.

Hvorfor sker dette? Tja, tænk på dynamikken i hukommelse og opmærksomhed i menneskelig kognition, som til en vis grad afspejles i store sprogmodeller. Når de behandler store mængder information, kan sprogmodeller, ligesom mennesker, miste overblikket over tidligere detaljer, mens de absorberer nye. Dette er især tilfældet i modeller, der ikke er eksplicit designet til automatisk at prioritere eller genbesøge tidligere tekstsegmenter.

Desuden er en sprogmodels evne til at væve disse udtrukne oplysninger sammen til et sammenhængende svar beslægtet med opbygning af narrativ. Dette kræver ikke kun udtrækning af information, men en dyb forståelse og kontekstuel placering, hvilket fortsat er en stor udfordring for nuværende kunstig intelligens.

Så hvad betyder dette for os som udviklere og integratorer af disse teknologier? Vi skal være meget opmærksomme på disse begrænsninger, når vi designer systemer, der er afhængige af store sprogmodeller til at håndtere komplekse opgaver med lange tekster. At forstå at ydeevnen kan forringes under visse forhold hjælper os med at sætte realistiske forventninger og udvikle bedre fallback-mekanismer eller supplerende strategier.

## Modaliteter: Ud over tekst

Mens størstedelen af sprogmodeller i dag fokuserer på at behandle og generere tekst, er der en voksende tendens mod multimodale modeller, der naturligt kan indlæse og outputte flere typer data, såsom billeder, lyd og video. Disse multimodale modeller åbner nye muligheder for AI-drevne applikationer, der kan forstå og generere indhold på tværs af forskellige modaliteter.

### Hvad er modaliteter?

I forbindelse med sprogmodeller refererer modaliteter til de forskellige typer data, som en model kan behandle og generere. Den mest almindelige modalitet er tekst, som omfatter skrevet sprog i forskellige former som bøger, artikler, hjemmesider og sociale medieindlæg. Der er dog flere andre modaliteter, som i stigende grad bliver inkorporeret i sprogmodeller:

- **Billeder**: Visuelle data såsom fotografier, illustrationer og diagrammer.
- **Lyd**: Lyddata såsom tale, musik og omgivelseslyde.
- **Video**: Bevægelige visuelle data, ofte ledsaget af lyd, såsom videoklip og film.

Hver modalitet præsenterer unikke udfordringer og muligheder for sprogmodeller. For eksempel kræver billeder, at modellen forstår visuelle koncepter og relationer, mens lyd kræver, at modellen behandler og genererer tale og andre lyde.

### Multimodale sprogmodeller

Multimodale sprogmodeller er designet til at håndtere flere modaliteter inden for en enkelt model. Disse modeller har typisk specialiserede komponenter eller lag, der både kan forstå input og generere output-data i forskellige modaliteter. Nogle bemærkelsesværdige eksempler på multimodale sprogmodeller omfatter:

- **OpenAI's GPT-4o**: GPT-4o er en stor sprogmodel, der naturligt forstår og behandler talelyd ud over tekst. Denne kapabilitet gør det muligt for GPT-4o at udføre opgaver såsom transskription af talt sprog, generering af tekst fra lydinput og levering af svar baseret på talte forespørgsler.
- **OpenAI's GPT-4 med visuelt input**: GPT-4 er en stor sprogmodel, der kan behandle både tekst og billeder. Når den får et billede som input, kan GPT-4 analysere billedets indhold og generere tekst, der beskriver eller reagerer på den visuelle information.
- **Google's Gemini**: Gemini er en multimodal model, der kan håndtere tekst, billeder og video. Den bruger en samlet arkitektur, der muliggør tværmodal forståelse og generering, hvilket muliggør opgaver som billedtekstning, videoopsummering og visuel spørgsmål-besvarelse.

- **DALL-E og Stable Diffusion**: Selvom disse ikke er sprogmodeller i traditionel forstand, demonstrerer de kraften i multimodal AI ved at generere billeder fra tekstbeskrivelser. De viser potentialet for modeller, der kan oversætte mellem forskellige modaliteter.

## Fordele og Anvendelser af Multimodale Modeller

Multimodale sprogmodeller tilbyder flere fordele og muliggør en bred vifte af anvendelser, herunder:

- **Forbedret forståelse**: Ved at behandle information fra flere modaliteter kan disse modeller opnå en mere omfattende forståelse af verden, lignende den måde mennesker lærer fra forskellige sensoriske inputs.
- **Krydsmodal generering**: Multimodale modeller kan generere indhold i én modalitet baseret på input fra en anden, såsom at skabe et billede fra en tekstbeskrivelse eller generere et videosammendrag fra en skreven artikel.

- **Tilgængelighed**: Multimodale modeller kan gøre information mere tilgængelig ved at oversætte mellem modaliteter, såsom at generere tekstbeskrivelser af billeder for synshandicappede brugere eller skabe lydversioner af skrevet indhold.
- **Kreative anvendelser**: Multimodale modeller kan bruges til kreative opgaver som at generere kunst, musik eller videoer baseret på tekstprompter, hvilket åbner nye muligheder for kunstnere og indholdskreatører.

Efterhånden som multimodale sprogmodeller fortsætter med at udvikle sig, vil de sandsynligvis spille en stadig vigtigere rolle i udviklingen af AI-drevne applikationer, der kan forstå og generere indhold på tværs af flere modaliteter. Dette vil muliggøre mere naturlige og intuitive interaktioner mellem mennesker og AI-systemer samt åbne for nye muligheder inden for kreativ udfoldelse og vidensformidling.

## Udbyder-økosystemer

Når det kommer til at inkorporere store sprogmodeller (LLMs) i applikationer, har du et voksende udvalg af muligheder at vælge imellem. Hver større LLM-udbyder, såsom OpenAI, Anthropic, Google og Cohere, tilbyder sit eget økosystem af modeller, API'er og værktøjer. At vælge den rigtige udbyder involverer overvejelse af forskellige faktorer, herunder prissætning, ydeevne, indholdsfiltrering, databeskyttelse og tilpasningsmuligheder.

### OpenAI

OpenAI er en af de mest velkendte udbydere af LLMs, hvor deres GPT-serie (GPT-3, GPT-4) bruges bredt i forskellige applikationer. OpenAI tilbyder et brugervenligt API, der gør det nemt at integrere deres modeller i applikationer. De tilbyder en række modeller med forskellige kapaciteter og prisniveauer, fra den grundlæggende Ada-model til den kraftfulde Davinci-model.

OpenAIs økosystem inkluderer også værktøjer som OpenAI Playground, der giver dig mulighed for at eksperimentere med prompts og finjustere modeller til specifikke

anvendelser. De tilbyder indholdsfiltrering for at hjælpe med at forhindre generering af upassende eller skadeligt indhold.

Når jeg bruger OpenAIs modeller direkte, benytter jeg Alex Rudalls ruby-openai bibliotek.

## Anthropic

Anthropic er en anden stor aktør inden for LLM-området, hvor deres Claude-modeller vinder popularitet for stærk ydeevne og etiske overvejelser. Anthropic fokuserer på at udvikle sikre og ansvarlige AI-systemer med stor vægt på indholdsfiltrering og undgåelse af skadelige outputs.

Anthropics økosystem omfatter Claude API'et, som giver dig mulighed for at integrere modellen i deres applikationer, samt værktøjer til prompt-udvikling og finjustering. De tilbyder også Claude Instant-modellen, som inkorporerer websøgning for mere opdaterede og faktuelle svar.

Når jeg bruger Anthropics modeller direkte, benytter jeg Alex Rudalls anthrophic bibliotek.

## Google

Google har udviklet flere kraftfulde LLMs, herunder Gemini, BERT, T5 og PaLM. Disse modeller er kendt for deres stærke præstationer inden for en bred vifte af opgaver inden for naturlig sprogbehandling. Googles økosystem omfatter TensorFlow- og Keras-bibliotekerne, som leverer værktøjer og rammer til at bygge og træne maskinlæringsmodeller.

Google tilbyder også en Cloud AI Platform, som gør det nemt at implementere og skalere deres modeller i skyen. De leverer en række prætrænede modeller og API'er til opgaver som sentimentanalyse, entitetsgenkendelse og oversættelse.

## Meta

Meta, tidligere kendt som Facebook, er dybt involveret i udviklingen af store sprogmodeller, hvilket understreges af deres frigivelse af modeller som LLaMA og OPT. Disse modeller udmærker sig ved deres stærke præstationer i forskellige sprogopgaver og er primært tilgængelige gennem open source-kanaler, hvilket understøtter Metas engagement i forskning og samarbejde med fællesskabet.

Metas økosystem er primært bygget omkring PyTorch, et open source-maskinlæringsbibliotek, der er foretrukket for dets dynamiske beregningsevner og fleksibilitet, hvilket faciliterer innovativ AI-forskning og -udvikling.

Ud over deres tekniske tilbud lægger Meta stor vægt på etisk AI-udvikling. De implementerer robust indholdsfiltrering og fokuserer på at reducere bias, hvilket stemmer overens med deres bredere mål om sikkerhed og ansvarlighed i AI-applikationer.

## Cohere

Cohere er en nyere aktør inden for LLM-området, der fokuserer på at gøre LLM'er mere tilgængelige og lettere at bruge end konkurrenterne. Deres økosystem inkluderer Cohere API'en, som giver adgang til en række præ-trænede modeller til opgaver som tekstgenerering, klassificering og opsummering.

Cohere tilbyder også værktøjer til prompt engineering, fine-tuning og indholdsfiltrering. De lægger vægt på databeskyttelse og sikkerhed med funktioner som krypteret datalagring og adgangskontrol.

## Ollama

Ollama er en selvhostet platform, der giver brugere mulighed for at administrere og implementere forskellige store sprogmodeller (LLM'er) lokalt på deres maskiner, hvilket

giver dem fuld kontrol over deres AI-modeller uden at være afhængige af eksterne cloud-tjenester. Denne opsætning er ideel for dem, der prioriterer databeskyttelse og ønsker at håndtere deres AI-operationer internt.

Platformen understøtter en række modeller, herunder versioner af Llama, Phi, Gemma og Mistral, som varierer i størrelse og beregningskrav. Ollama gør det nemt at downloade og køre disse modeller direkte fra kommandolinjen ved hjælp af simple kommandoer som `ollama run <model_name>`, og den er designet til at fungere på tværs af forskellige operativsystemer, herunder macOS, Linux og Windows.

For udviklere, der ønsker at integrere open source-modeller i deres applikationer uden at bruge et eksternt API, tilbyder Ollama en CLI til håndtering af modellers livscyklus, der minder om værktøjer til containerhåndtering. Den understøtter også brugerdefinerede konfigurationer og prompts, hvilket giver mulighed for en høj grad af tilpasning til specifikke behov eller anvendelser.

Ollama er særligt velegnet til teknisk kyndige brugere og udviklere på grund af dens kommandolinjeinterface og den fleksibilitet, den tilbyder i forhold til at administrere og implementere AI-modeller. Dette gør det til et kraftfuldt værktøj for virksomheder og enkeltpersoner, der har behov for robuste AI-funktioner uden at gå på kompromis med sikkerhed og kontrol.

### Multi-model-platforme

Derudover findes der udbydere, der hoster en bred vifte af open source-modeller, såsom Together.ai og Groq. Disse platforme tilbyder fleksibilitet og tilpasningsmuligheder, der giver dig mulighed for at køre og i nogle tilfælde endda fine-tune open source-modeller efter dine specifikke behov. For eksempel giver Together.ai adgang til en række open source LLM'er, hvilket giver brugerne mulighed for at eksperimentere med forskellige modeller og konfigurationer. Groq fokuserer på at levere ultrahøj ydeevne i færdiggørelsen, som på tidspunktet for denne bogs udgivelse virker næsten magisk

## Valg af LLM-udbyder

Når du vælger en LLM-udbyder, bør du overveje faktorer som:

- **Prissætning**: Forskellige udbydere tilbyder forskellige prismodeller, lige fra betaling pr. brug til abonnementsbaserede planer. Det er vigtigt at overveje det forventede forbrug og budget, når man vælger en udbyder.
- **Ydeevne**: LLM'ers ydeevne kan variere betydeligt mellem udbydere, så det er vigtigt at benchmarke og teste modeller på specifikke anvendelser, før man træffer en beslutning.
- **Indholdsfiltrering**: Afhængigt af anvendelsen kan indholdsfiltrering være en kritisk overvejelse. Nogle udbydere tilbyder mere robuste indholdsfiltreringsmuligheder end andre.
- **Databeskyttelse**: Hvis applikationen håndterer følsomme brugerdata, er det vigtigt at vælge en udbyder med stærk databeskyttelse og sikkerhedspraksis.
- **Tilpasning**: Nogle udbydere tilbyder mere fleksibilitet med hensyn til fine-tuning og tilpasning af modeller til specifikke anvendelser.

I sidste ende afhænger valget af LLM-udbyder af applikationens specifikke krav og begrænsninger. Ved omhyggeligt at evaluere mulighederne og overveje faktorer som prissætning, ydeevne og databeskyttelse kan du vælge den udbyder, der bedst opfylder dine behov.

Det er også værd at bemærke, at LLM-landskabet konstant udvikler sig, og nye udbydere og modeller dukker regelmæssigt op. Du bør holde dig opdateret med den seneste udvikling og være åben for at udforske nye muligheder, efterhånden som de bliver tilgængelige.

## OpenRouter

Gennem denne bog vil jeg udelukkende bruge OpenRouter som min foretrukne API-udbyder. Årsagen er enkel: det er en one-stop-shop for alle de mest populære

kommercielle og open source-modeller. Hvis du er ivrig efter at komme i gang med noget AI-kodning, er et af de bedste steder at starte med mit eget OpenRouter Ruby Library.

## Om Ydeevne

Når man indbygger sprogmodeller i applikationer, er ydeevne en kritisk overvejelse. En sprogmodels ydeevne kan måles i form af dens *latens* (den tid det tager at generere et svar) og *gennemløb* (antallet af forespørgsler den kan håndtere pr. tidsenhed).

*Tid til første token* (TTFT) er endnu et væsentligt ydeevnemål, særligt relevant for chatbots og applikationer der kræver interaktive svar i realtid. TTFT måler latenstiden fra det øjeblik en brugers forespørgsel modtages, til det første ord (eller token) i svaret genereres. Dette mål er afgørende for at opretholde en problemfri og engagerende brugeroplevelse, da forsinkede svar kan føre til brugerfrustrationer og manglende engagement.

Disse ydelsesmål kan have betydelig indflydelse på brugeroplevelsen og applikationens skalerbarhed.

Flere faktorer kan påvirke en sprogmodels ydeevne, herunder:

**Parameterantal**: Større modeller med flere parametre kræver generelt flere computerressourcer og kan have højere latens og lavere gennemløb sammenlignet med mindre modeller.

**Hardware**: En sprogmodels ydeevne kan variere betydeligt afhængigt af den hardware, den kører på. Cloud-udbydere tilbyder GPU- og TPU-instanser optimeret til maskinlæringsworkloads, hvilket kan accelerere modelinferens betydeligt.

En af de fine ting ved OpenRouter er, at for mange af de modeller den tilbyder, får du valget mellem cloud-udbydere med forskellige ydeevneprofiler og omkostninger.

**Kvantisering**: Kvantiseringsteknikker kan bruges til at reducere en models hukommelsesforbrug og beregningskrav ved at repræsentere vægte og aktiveringer med datatyper af lavere præcision. Dette kan forbedre ydeevnen uden at ofre kvaliteten væsentligt. Som applikationsudvikler vil du sandsynligvis ikke blive involveret i træning af dine egne modeller med forskellige kvantiseringsniveauer, men det er godt at være fortrolig med terminologien.

**Batchprocessering**: Behandling af flere forespørgsler samtidigt i batches kan forbedre gennemløbet ved at amortisere overhead fra modelindlæsning og dataoverførsel.

**Cachelagring**: Cachelagring af resultater fra hyppigt anvendte prompts eller inputsekvenser kan reducere antallet af inferensforespørgsler og forbedre den generelle ydeevne.

Når man vælger en sprogmodel til en produktionsapplikation, er det vigtigt at benchmarke dens ydeevne på repræsentative workloads og hardwarekonfigurationer. Dette kan hjælpe med at identificere potentielle flaskehalse og sikre, at modellen kan opfylde de krævede ydelsesmål.

Det er også værd at overveje afvejningerne mellem modelydeevne og andre faktorer som omkostninger, fleksibilitet og integrationsvenlighed. For eksempel kan brugen af en mindre, billigere model med lavere latens være at foretrække for applikationer, der kræver realtidssvar, mens en større, mere kraftfuld model kan være bedre egnet til batchprocessering eller komplekse ræsonnementsopgaver.

## Eksperimenter Med Forskellige LLM-Modeller

At vælge en LLM er sjældent en permanent beslutning. Da nye og forbedrede modeller udgives regelmæssigt, er det godt at bygge applikationer på en modulær måde, der tillader udskiftning af forskellige sprogmodeller over tid. Prompts og datasæt kan ofte genbruges på tværs af modeller med minimale ændringer. Dette giver dig mulighed for

at udnytte de seneste fremskridt inden for sprogmodellering uden at skulle redesigne dine applikationer fuldstændigt.

Muligheden for nemt at skifte mellem et bredt udvalg af modelvalg er endnu en grund til, at jeg elsker OpenRouter.

Når man opgraderer til en ny sprogmodel, er det vigtigt grundigt at teste og validere dens ydeevne og outputkvalitet for at sikre, at den opfylder applikationens krav. Dette kan involvere gentræning eller finjustering af modellen på domænespecifikke data, samt opdatering af eventuelle downstream-komponenter, der afhænger af modellens output.

Ved at designe applikationer med fokus på ydeevne og modularitet kan du skabe skalerbare, effektive og fremtidssikrede systemer, der kan tilpasse sig det hurtigt udviklende landskab inden for sprogmodelleringsteknologi.

## Sammensatte AI-Systemer

Før vi afslutter vores introduktion, er det værd at nævne, at før 2023 og eksplosionen i interessen for generativ AI udløst af ChatGPT, var traditionelle AI-tilgange typisk afhængige af integration af enkelte, lukkede modeller. I modsætning hertil udnytter *Sammensatte AI-Systemer* komplekse pipelines af sammenkoblede komponenter, der arbejder sammen om at opnå intelligent adfærd.

I kernen består sammensatte AI-systemer af flere moduler, der hver er designet til at udføre specifikke opgaver eller funktioner. Disse moduler kan omfatte generatorer, hentningskomponenter, rangeringskomponenter, klassifikatorer og forskellige andre specialiserede komponenter. Ved at opdele det samlede system i mindre, fokuserede enheder kan udviklere skabe mere fleksible, skalerbare og vedligeholdelsesvenlige AI-arkitekturer.

En af de vigtigste fordele ved sammensatte AI-systemer er deres evne til at kombinere styrkerne fra forskellige AI-teknikker og modeller. For eksempel kan et system bruge en

stort sprogmodel (LLM) til forståelse og generering af naturligt sprog, mens det anvender en separat model til informationssøgning eller regelbaseret beslutningstagning. Denne modulære tilgang giver dig mulighed for at vælge de bedste værktøjer og teknikker til hver specifik opgave, frem for at være afhængig af en universalløsning.

Dog præsenterer opbygningen af sammensatte AI-systemer også unikke udfordringer. Særligt kræver sikring af systemets overordnede sammenhæng og konsistens robuste test-, overvågnings- og styringsmekanismer.

Fremkomsten af kraftfulde LLM'er som GPT-4 gør det lettere end nogensinde før at eksperimentere med sammensatte AI-systemer, fordi disse avancerede modeller er i stand til at håndtere flere roller inden for et sammensat system, såsom klassificering, rangering og generering, ud over deres evner til at forstå naturligt sprog. Denne alsidighed gør det muligt for udviklere hurtigt at udvikle prototyper og iterere på sammensatte AI-arkitekturer, hvilket åbner nye muligheder for udvikling af intelligente applikationer.

## Implementeringsmønstre for Sammensatte AI-systemer

Sammensatte AI-systemer kan implementeres ved hjælp af forskellige mønstre, der hver er designet til at imødekomme specifikke krav og anvendelsesområder. Lad os udforske fire almindelige implementeringsmønstre: Spørgsmål og Svar, Multi-Agent/Agentiske Problemløsere, Konversations-AI og CoPilots.

### Spørgsmål og Svar

Spørgsmål og Svar (Q&A) systemer fokuserer på at levere informationssøgning, der er forbedret med AI-modellers forståelsesevner for at fungere som mere end blot en søgemaskine. Ved at kombinere kraftfulde sprogmodeller med eksterne videnskilder ved hjælp af Genfindelses-forstærket Generering (RAG), undgår Spørgsmål og

Svar-systemer hallucinationer og giver præcise og kontekstuelt relevante svar på brugerforespørgsler.

De vigtigste komponenter i et LLM-baseret Q&A-system omfatter:

- **Forespørgselsforståelse og -omformulering**: Analyse af brugerforespørgsler og omformulering af disse for bedre at matche de underliggende videnskilder.
- **Vidensgenfinding**: Genfinding af relevant information fra strukturerede eller ustrukturerede datakilder baseret på den omformulerede forespørgsel.
- **Svargenerering**: Generering af sammenhængende og informative svar ved at integrere den genfundne viden med sprogmodellens generative evner.

RAG-delsystemer er særligt vigtige i Q&A-domæner, hvor det er afgørende at levere præcis og opdateret information, såsom kundesupport, vidensstyring eller uddannelsesapplikationer

## Multi-Agent/Agentiske Problemløsere

Multi-agent, også kendt som *Agentiske*, systemer består af flere autonome agenter, der arbejder sammen om at løse komplekse problemer. Hver agent har en specifik rolle, et sæt færdigheder og adgang til relevante værktøjer eller informationskilder. Ved at samarbejde og udveksle information kan disse agenter tackle opgaver, som ville være vanskelige eller umulige for en enkelt agent at håndtere alene.

De vigtigste principper for multi-agent problemløsere omfatter:

- **Specialisering**: Hver agent fokuserer på et specifikt aspekt af problemet og udnytter sine unikke evner og viden.
- **Samarbejde**: Agenter kommunikerer og koordinerer deres handlinger for at nå et fælles mål, ofte gennem beskeder eller delt hukommelse.
- **Tilpasningsevne**: Systemet kan tilpasse sig ændrede forhold eller krav ved at justere de enkelte agenters roller og adfærd.

Multi-agent systemer er velegnede til applikationer, der kræver distribueret problemløsning, såsom forsyningskædeoptimering, trafikstyring eller planlægning af beredskab

## Konversations-AI

Konversations-AI-systemer muliggør interaktioner på naturligt sprog mellem brugere og intelligente agenter. Disse systemer kombinerer forståelse af naturligt sprog, dialoghåndtering og sproggenereringsevner for at levere engagerende og personlige samtalebaserede oplevelser.

Hovedkomponenterne i et konversations-AI-system omfatter:

- **Intentionsgenkendelse**: Identificering af brugerens intention baseret på deres input, såsom at stille et spørgsmål, fremsætte en anmodning eller udtrykke en følelse.
- **Entitetsudtrækning**: Udtrækning af relevante entiteter eller parametre fra brugerens input, såsom datoer, lokationer eller produktnavne.
- **Dialoghåndtering**: Vedligeholdelse af samtalens tilstand, bestemmelse af passende svar baseret på brugerens intention og kontekst, samt håndtering af flerturs-interaktioner.
- **Svargenerering**: Generering af menneskelignende svar ved hjælp af sprogmodeller, skabeloner eller genfindelsesbaserede metoder.

Konversations-AI-systemer bruges almindeligvis i kundeservice-chatbots, virtuelle assistenter og stemmestyrede grænseflader. Som nævnt tidligere er de fleste af tilgangene, mønstrene og kodeeksemplerne i denne bog direkte uddraget fra mit arbejde med et stort konversations-AI-system kaldet Olympia

## CoPilots

CoPilots er AI-drevne assistenter, der arbejder sammen med menneskelige brugere for at forbedre deres produktivitet og beslutningstagning. Disse systemer udnytter en

kombination af naturlig sprogbehandling, maskinlæring og domænespecifik viden til at give intelligente anbefalinger, automatisere opgaver og tilbyde kontekstuel støtte.

Centrale funktioner i CoPilots omfatter:

- **Personalisering**: Tilpasning til individuelle brugerpræferencer, arbejdsgange og kommunikationsstile.
- **Proaktiv assistance**: Foregribelse af brugerens behov og tilbud om relevante forslag eller handlinger uden eksplicitte forespørgsler.
- **Kontinuerlig læring**: Forbedring af ydeevne over tid gennem læring fra brugerfeedback, interaktioner og data.

CoPilots bruges i stigende grad inden for forskellige domæner, såsom softwareudvikling (f.eks. kodekomplettering og fejlfinding), kreativ skrivning (f.eks. indholdsforslag og redigering), og dataanalyse (f.eks. indsigter og visualiseringsanbefalinger)

Disse implementeringsmønstre viser alsidigheden og potentialet i sammensatte AI-systemer. Ved at forstå karakteristikaene og anvendelsesmulighederne for hvert mønster kan du træffe informerede beslutninger ved design og implementering af intelligente applikationer. Selvom denne bog ikke specifikt handler om implementering af sammensatte AI-systemer, gælder mange, hvis ikke alle, af de samme tilgange og mønstre for integration af diskrete AI-komponenter inden for ellers traditionel applikationsudvikling.

## Roller i sammensatte AI-systemer

Sammensatte AI-systemer er bygget på et fundament af sammenkoblede moduler, der hver er designet til at udføre en specifik rolle. Disse moduler arbejder sammen om at skabe intelligent adfærd og løse komplekse problemer. Det er nyttigt at være fortrolig med disse roller, når man overvejer, hvor man kunne implementere eller erstatte dele af sin applikation med diskrete AI-komponenter.

## Generator

Generatorer er ansvarlige for at producere nye data eller indhold baseret på lærte mønstre eller input-prompts. AI-verdenen har mange forskellige slags generatorer, men i forbindelse med de sprogmodeller, der præsenteres i denne bog, kan generatorer skabe menneskelignende tekst, fuldføre delvise sætninger eller generere svar på brugerforespørgsler. De spiller en afgørende rolle i opgaver som indholdsproduktion, dialoggenerering og dataforøgelse.

## Informationshenter

Informationshentere bruges til at søge og udtrække relevant information fra store datasæt eller videnbaser. De anvender teknikker som semantisk søgning, nøgleordsmatchning eller vektorsimilaritet til at finde de mest relevante datapunkter baseret på en given forespørgsel eller kontekst. Informationshentere er essentielle for opgaver, der kræver hurtig adgang til specifik information, såsom besvarelse af spørgsmål, faktakontrol eller indholdsanbefaling.

## Rangordner

Rangordnere er ansvarlige for at ordne eller prioritere et sæt elementer baseret på bestemte kriterier eller relevansscorer. De tildeler vægte eller scorer til hvert element og sorterer dem derefter i overensstemmelse hermed. Rangordnere bruges almindeligvis i søgemaskiner, anbefalingssystemer eller enhver applikation, hvor præsentation af de mest relevante resultater for brugerne er afgørende.

## Klassifikator

Klassifikatorer bruges til at kategorisere eller mærke datapunkter baseret på foruddefinerede klasser eller kategorier. De lærer fra mærket træningsdata og forudsiger derefter klassen for nye, usete tilfælde. Klassifikatorer er fundamentale for

opgaver som sentimentanalyse, spam-detektion eller billedgenkendelse, hvor målet er at tildele en specifik kategori til hvert input.

## Værktøjer & Agenter

Ud over disse kerneroller inkorporerer sammensatte AI-systemer ofte værktøjer og agenter for at forbedre deres funktionalitet og tilpasningsevne:

- **Værktøjer**: Værktøjer er diskrete softwarekomponenter eller API'er, der udfører specifikke handlinger eller beregninger. De kan kaldes af andre moduler, såsom generatorer eller informationshentere, for at udføre delopgaver eller indsamle yderligere information. Eksempler på værktøjer omfatter websøgemaskiner, lommeregnere eller datavisualiseringsbiblioteker.
- **Agenter**: Agenter er autonome enheder, der kan opfatte deres omgivelser, træffe beslutninger og handle for at opnå specifikke mål. De er ofte afhængige af en kombination af forskellige AI-teknikker, såsom planlægning, ræsonnement og læring, for at fungere effektivt under dynamiske eller usikre forhold. Agenter kan bruges til at modellere kompleks adfærd eller til at koordinere handlinger mellem flere moduler i et sammensat AI-system.

I et rent sammensat AI-system orchestreres interaktionen mellem disse komponenter gennem veldefinerede grænseflader og kommunikationsprotokoller. Data flyder mellem moduler, hvor output fra én komponent fungerer som input for en anden. Denne modulære arkitektur muliggør fleksibilitet, skalerbarhed og vedligeholdelse, da individuelle komponenter kan opdateres, erstattes eller udvides uden at påvirke hele systemet.

Ved at udnytte styrken i disse komponenter og deres interaktioner kan sammensatte AI-systemer tackle komplekse, virkelige problemer, der kræver en kombination af forskellige AI-kapabiliteter. Mens vi udforsker tilgangene og mønstrene for integration af AI i applikationsudvikling, skal du huske på, at de samme principper og teknikker,

der bruges i sammensatte AI-systemer, kan anvendes til at skabe intelligente, adaptive og brugercentrerede applikationer.

I de følgende kapitler i Del 1 vil vi dykke dybere ned i de fundamentale tilgange og teknikker til integration af AI-komponenter i din applikationsudviklingsproces. Fra prompt-udvikling og retrieval-augmented generation til selvhelende data og intelligent workflow-orkestrering vil vi dække en bred vifte af mønstre og best practices for at hjælpe dig med at bygge banebrydende AI-drevne applikationer.

# Del 1: Grundlæggende Tilgange & Teknikker

Denne del af bogen præsenterer forskellige måder at integrere brugen af AI i dine applikationer. Kapitlerne dækker en række beslægtede tilgange og teknikker, der spænder fra de mere overordnede koncepter som Indsnævre Stien og Retrieval Augmented Generation helt ned til idéer om at programmere dit eget abstraktionslag oven på LLM chat-færdiggørelses-API'er.

Målet med denne del af bogen er at hjælpe dig med at forstå de forskellige former for adfærd, du kan implementere med AI, før vi går for dybt ind i specifikke implementeringsmønstre, som er fokus i Del 2.

Tilgangene i Del 1 er baseret på idéer, som jeg har brugt i min kode, klassiske mønstre inden for virksomhedsapplikationsarkitektur og integration, plus metaforer, som jeg har brugt, når jeg har skullet forklare AI's muligheder til andre mennesker, herunder ikke-tekniske forretningsinteressenter.

# Indsnævre Stien

“Indsnævre stien” henviser til at fokusere AI’en på den opgave, der skal løses. Jeg bruger det som et mantra, når jeg bliver frustreret over, at AI’en opfører sig “dumt” eller på uventede måder. Mantraet minder mig om, at fejlen sandsynligvis er min egen, og at jeg formentlig bør indsnævre stien noget mere.

Behovet for at indsnævre stien opstår fra den enorme mængde viden, der findes i store sprogmodeller, især verdensklassemodeller som dem fra OpenAI og Anthropic, der bogstaveligt talt har billioner af parametre.

At have adgang til sådan et bredt spektrum af viden er uden tvivl kraftfuldt og producerer emergent adfærd såsom theory of mind og evnen til at ræsonnere på menneskelig vis. Denne skelsættende mængde information skaber dog også udfordringer, når det kommer til at generere præcise og nøjagtige svar på specifikke prompts, især hvis disse prompts skal udvise deterministisk adfærd, der kan integreres med "normal" softwareudvikling og algoritmer.

En række faktorer fører til disse udfordringer.

**Informationsoverbelastning**: Store sprogmodeller er trænet på massive mængder data, der spænder over forskellige domæner, kilder og tidsperioder. Denne omfattende viden gør dem i stand til at engagere sig i forskellige emner og generere svar baseret på en bred forståelse af verden. Når modellen står over for et specifikt prompt, kan den dog kæmpe med at filtrere irrelevant, modstridende eller forældet/obsolet information fra, hvilket fører til svar, der mangler fokus eller præcision. Afhængigt af hvad du forsøger at gøre, kan den rene mængde af *modstridende* information, der er tilgængelig for modellen, let overvælde dens evne til at give det svar eller den adfærd, du søger.

**Kontekstuel Tvetydighed**: I betragtning af det enorme *latente rum* af viden kan store sprogmodeller støde på tvetydighed, når de forsøger at forstå *konteksten* af dit prompt. Uden ordentlig indsnævring eller vejledning kan modellen generere svar, der kun er perifert relaterede, men ikke direkte relevante for dine intentioner. Denne type fejl fører til svar, der er uden for emnet, inkonsistente eller ikke imødekommer dine angivne behov. I dette tilfælde henviser indsnævring af stien til kontekst *afklaring*, der sikrer, at den kontekst, du giver, får modellen til kun at fokusere på den mest relevante information i dens grundlæggende viden.

Bemærk: Når du starter med "prompt engineering", er du meget mere tilbøjelig til at bede modellen om at gøre ting uden at forklare det ønskede resultat ordentligt; det kræver øvelse ikke at være tvetydig!

**Tidsmæssige Uoverensstemmelser**: Da sprogmodeller er trænet på data, der blev skabt

på forskellige tidspunkter, kan de besidde viden, der er forældet, erstattet eller ikke længere præcis. For eksempel kan information om aktuelle begivenheder, videnskabelige opdagelser eller teknologiske fremskridt have udviklet sig siden modellens træningsdata blev indsamlet. Uden at indsnævre stien til at prioritere nyere og mere pålidelige kilder kan modellen generere svar baseret på forældet eller ukorrekt information, hvilket fører til unøjagtigheder og inkonsistens i dens output.

**Domænespecifikke Nuancer**: Forskellige domæner og felter har deres egen specifikke terminologi, konventioner og vidensbase. Tænk på stort set enhver TLA (Three Letter Acronym), og du vil indse, at de fleste af dem har mere end én betydning. For eksempel kan MSK henvise til Amazon's Managed Streaming for Apache Kafka, Memorial Sloan Kettering Cancer Center eller det menneskelige muskuloskeletale system.

Når et prompt kræver ekspertise inden for et bestemt domæne, er en stor sprogmodels generiske viden måske ikke tilstrækkelig til at give præcise og nuancerede svar. At indsnævre stien ved at fokusere på domænespecifik information, enten gennem prompt engineering eller retrieval-augmented generation, gør det muligt for modellen at generere svar, der er mere på linje med dit specifikke domænes krav og forventninger.

## Latent Rum: Ubegribeligt Stort

Når jeg nævner "latent rum" i en sprogmodel, henviser jeg til det enorme, multidimensionelle landskab af viden og information, som modellen har lært under sin træningsproces. Det er som et skjult rige inden i modellens neurale netværk, hvor alle mønstre, associationer og repræsentationer af sprog er gemt.

Forestil dig, at du udforsker et stort, ukendt territorium fyldt med utallige sammenkoblede knudepunkter. Hvert knudepunkt repræsenterer et stykke information, et koncept eller en relation, som modellen har lært. Når du navigerer gennem dette rum, vil du opdage, at nogle knudepunkter er tættere på hinanden, hvilket indikerer en stærk forbindelse eller lighed, mens andre er længere fra hinanden, hvilket antyder en svagere eller mere fjern relation.

Udfordringen med det latente rum er, at det er utroligt komplekst og højdimensionelt. Tænk på det som værende lige så enormt som vores fysiske univers, med dets galaksehobe og enorme, ufattelige afstande af tomt rum imellem dem.

Fordi det indeholder tusindvis af dimensioner, er det latente rum ikke direkte observerbart eller fortolkeligt for mennesker. Det er en abstrakt repræsentation, som modellen bruger internt til at behandle og generere sprog. Når du giver modellen et input-prompt, kortlægger den i princippet dette prompt til en specifik placering i det latente rum. Modellen bruger derefter den omkringliggende information og forbindelser i dette rum til at generere et svar.

Sagen er, at modellen har lært en enorm mængde information fra sine træningsdata, og ikke alt er relevant eller præcist for en given opgave. Det er derfor, indsnævring af stien bliver så vigtig. Ved at give klare instruktioner, eksempler og kontekst i dine prompts, guider du i realiteten modellen til at fokusere på specifikke områder inden for det latente rum, som er mest relevante for dit ønskede output.

En anden måde at tænke på det er som at bruge en spotlight i et helt mørkt museum. Hvis du nogensinde har besøgt Louvre eller Metropolitan Museum of Art, så er det den slags skala, jeg taler om. Det latente rum er museet, fyldt med utallige genstande og detaljer. Dit prompt er spotlightet, der oplyser specifikke områder og leder modellens opmærksomhed hen på den vigtigste information. Uden denne vejledning kan modellen vandre formålsløst gennem det latente rum og samle irrelevant eller modstridende information op undervejs.

Når du arbejder med sprogmodeller og udformer dine prompts, så husk konceptet om det latente rum. Dit mål er at navigere effektivt gennem dette enorme videnslandskab og styre modellen mod den mest relevante og præcise information til din opgave. Ved at indsnævre stien og give klar vejledning kan du frigøre det fulde potentiale i modellens latente rum og generere sammenhængende svar af høj kvalitet.

Mens de tidligere beskrivelser af sprogmodeller og det latente rum, de navigerer i, kan virke lidt magiske eller abstrakte, er det vigtigt at forstå, at prompts ikke er

trylleformularer eller besværgelser. Måden sprogmodeller fungerer på er forankret i principperne om lineær algebra og sandsynlighedsteori.

I deres kerne er sprogmodeller probabilistiske modeller af tekst, meget ligesom hvordan en normalfordelingskurve er en statistisk model af data. De trænes gennem en proces kaldet autoregressiv modellering, hvor modellen lærer at forudsige sandsynligheden for det næste ord i en sekvens baseret på de ord, der kommer før det. Under træningen starter modellen med tilfældige vægte og justerer dem gradvist for at tildele højere sandsynligheder til tekst, der ligner de virkelige eksempler, den blev trænet på.

Men at tænke på sprogmodeller som simple statistiske modeller, som lineær regression, giver ikke den bedste intuition for at forstå deres adfærd. En mere passende analogi er at tænke på dem som probabilistiske programmer, som er modeller der tillader manipulation af tilfældige variabler og kan repræsentere komplekse statistiske relationer.

Probabilistiske programmer kan repræsenteres af grafiske modeller, som giver en visuel måde at forstå afhængigheder og relationer mellem variabler i modellen. Dette perspektiv kan give værdifuld indsigt i funktionen af komplekse tekstgenererende modeller som GPT-4 og Claude.

I artiklen "Language Model Cascades" af Dohan et al. dykker forfatterne ned i detaljerne om, hvordan probabilistiske programmer kan anvendes på sprogmodeller. De viser, hvordan denne ramme kan bruges til at forstå disse modellers adfærd og guide udviklingen af mere effektive promptning-strategier.

En central indsigt fra dette probabilistiske perspektiv er, at sprogmodellen i det væsentlige skaber en portal til et alternativt univers, hvor de ønskede dokumenter eksisterer. Modellen tildeler vægte til alle mulige dokumenter baseret på deres sandsynlighed og indsnævrer effektivt rummet af muligheder for at fokusere på de mest relevante.

Dette bringer os tilbage til det centrale tema om "at indsnævre stien." Det primære mål med promptning er at betinge den probabilistiske model på en måde, der fokuserer

massen af dens forudsigelser og indsnævrer det til den specifikke information eller adfærd, vi ønsker at fremkalde. Ved at give omhyggeligt udformede prompts kan vi guide modellen til at navigere det latente rum mere effektivt og generere output, der er mere relevante og sammenhængende.

Det er dog vigtigt at huske, at sprogmodellen i sidste ende er begrænset af den information, den blev trænet på. Mens den kan generere tekst, der ligner eksisterende dokumenter eller kombinere idéer på nye måder, kan den ikke fremtrylle helt ny information ud af det blå. For eksempel kan vi ikke forvente, at modellen kan give en kur mod kræft, hvis en sådan kur ikke er blevet opdaget og dokumenteret i dens træningsdata.

I stedet ligger modellens styrke i dens evne til at finde og syntetisere information, der ligner det, vi prompter den med. Ved at forstå disse modellers probabilistiske natur og hvordan prompts kan bruges til at betinge deres output, kan vi mere effektivt udnytte deres evner til at generere værdifuld indsigt og indhold.

Overvej følgende prompts. I den første kunne "Mercury" alene henvise til planeten, grundstoffet eller den romerske gud, men det mest sandsynlige er planeten. GPT-4 giver faktisk et langt svar, der begynder med *Merkur er den mindste og inderste planet i solsystemet…*. Den anden prompt henviser specifikt til grundstoffet. Den tredje henviser til den romerske mytologiske figur, kendt for sin hastighed 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 at tilføje blot en håndfuld ekstra ord har vi fuldstændigt ændret, hvordan AI'en reagerer. Som du vil lære senere i bogen, er avancerede prompt-engineering-teknikker som n-shot prompting, struktureret input/output og tankerække (Chain of Thought) blot smarte måder at betinge modellens output på.

Så i sidste ende handler kunsten at lave prompt-engineering om at forstå, hvordan man navigerer i sprogmodellens omfattende probabilistiske videnslandskab for at indsnævre stien til den specifikke information eller adfærd, vi søger.

For læsere med en solid forståelse af avanceret matematik kan det bestemt hjælpe at basere din forståelse af disse modeller på principperne inden for sandsynlighedsteori og lineær algebra! For resten af jer, der ønsker at udvikle effektive strategier til at fremkalde ønskede outputs, lad os holde os til mere intuitive tilgange.

## Hvordan Stien Bliver "Indsnævret"

For at håndtere disse udfordringer med for meget viden anvender vi teknikker, der hjælper med at guide sprogmodellens genereringsproces og fokusere dens opmærksomhed på den mest relevante og præcise information.

Her er de vigtigste teknikker i anbefalet rækkefølge, det vil sige, du bør først prøve Prompt Engineering, derefter RAG, og til sidst, hvis det er nødvendigt, fin-tuning.

**Prompt Engineering** Den mest grundlæggende tilgang er at udforme prompts, der inkluderer specifikke instruktioner, begrænsninger eller eksempler til at guide modellens

responsgenerering. Dette kapitel dækker grundprincipperne i Prompt Engineering i næste afsnit, og vi dækker mange specifikke prompt-engineering-mønstre i Del 2 af bogen. Disse mønstre inkluderer Prompt-destillering, en teknik der fokuserer på at forfine og optimere prompts for at udtrække det, som AI'en anser for at være den mest relevante og præcise information.

**Kontekstudvidelse**. Dynamisk hentning af relevant information fra eksterne vidensbasere eller dokumenter for at forsyne modellen med fokuseret kontekst på det tidspunkt, hvor den promptes. Populære kontekstudvidelsesteknikker inkluderer Retrieval-Augmented Generation (RAG) Såkaldte "online-modeller" som dem, der leveres af Perplexity, er i stand til at udvide deres kontekst med realtids internetsøgeresultater.

På trods af deres kraft er LLM'er ikke trænet på dine unikke datasæt, som kan være private eller specifikke for det problem, du forsøger at løse. Kontekstudvidelsesteknikker lader dig give LLM'er adgang til data bag API'er, i SQL-databaser eller fanget i PDF'er og præsentationer.

**Fin-tuning eller domænetilpasning** Træning af modellen på domænespecifikke datasæt for at specialisere dens viden og genereringsevner til en bestemt opgave eller felt.

## At Skrue Ned For Temperaturen

Temperatur er en *hyperparameter*, der bruges i transformer-baserede sprogmodeller til at kontrollere tilfældigheden og kreativiteten i den genererede tekst. Det er en værdi mellem 0 og 1, hvor lavere værdier gør outputtet mere fokuseret og deterministisk, mens højere værdier gør det mere mangfoldigt og uforudsigeligt.

Når temperaturen er sat til 1, genererer sprogmodellen tekst baseret på den fulde sandsynlighedsfordeling for det næste token, hvilket tillader mere kreative og varierede

svar. Dette kan dog også føre til, at modellen genererer tekst, der er mindre relevant eller sammenhængende.

På den anden side, når temperaturen er sat til 0, vælger sprogmodellen altid det token med den højeste sandsynlighed, hvilket effektivt "indsnævrer dens sti." Næsten alle mine AI-komponenter bruger en temperatur sat på eller tæt på 0, da det resulterer i mere fokuserede og forudsigelige svar. Det er absolut nyttigt, når du vil have modellen til at *følge instruktioner*, være opmærksom på funktioner, den har fået stillet til rådighed, eller simpelthen har brug for mere præcise og relevante svar end det, du får.

For eksempel, hvis du bygger en chatbot, der skal levere faktuel information, vil du måske indstille temperaturen til en lavere værdi for at sikre, at svarene er mere præcise og relevante. Omvendt, hvis du bygger en kreativ skriveassistent, vil du måske indstille temperaturen til en højere værdi for at fremme mere mangfoldige og fantasifulde outputs.

## Hyperparametre: Inferensens Knapper og Drejehjul

Når du arbejder med sprogmodeller, vil du ofte støde på begrebet "hyperparametre". I forbindelse med inferens (dvs. når du bruger modellen til at generere svar) er hyperparametre som knapper og drejehjul, du kan justere for at kontrollere modellens adfærd og output.

Tænk på det som at justere indstillingerne på en kompleks maskine. Ligesom du måske drejer på en knap for at kontrollere temperaturen eller skifter en kontakt for at ændre driftsmåden, giver hyperparametre dig mulighed for at finjustere den måde, sprogmodellen behandler og genererer tekst på.

Nogle almindelige hyperparametre, du vil støde på under inferens, omfatter:

- **Temperatur**: Som lige nævnt styrer denne parameter tilfældigheden og kreativiteten i den genererede tekst. En højere temperatur fører til mere

forskelligartede og uforudsigelige outputs, mens en lavere temperatur resulterer i mere fokuserede og deterministiske svar.

- **Top-p (nucleus) sampling**: Denne parameter styrer udvælgelsen af det mindste sæt tokens, hvis kumulative sandsynlighed overstiger en bestemt tærskel (p). Det muliggør mere forskelligartede outputs, samtidig med at sammenhængen bevares.

- **Top-k sampling**: Denne teknik vælger de *k* mest sandsynlige næste tokens og omfordeler sandsynlighedsmassen mellem dem. Det kan hjælpe med at forhindre modellen i at generere tokens med lav sandsynlighed eller irrelevante tokens.

- **Frekvens-** og **Tilstedeværelsesstraf**: Disse parametre straffer modellen for at gentage de samme ord eller sætninger for ofte (frekvensstraf) eller for at generere ord, der ikke er til stede i input-prompten (tilstedeværelsesstraf). Ved at justere disse værdier kan du få modellen til at producere mere varierede og relevante outputs.

- **Maksimal længde**: Denne hyperparameter sætter en øvre grænse for antallet af tokens (ord eller delord), som modellen kan generere i et enkelt svar. Det hjælper med at kontrollere ordrigeligheden og præcisionen af den genererede tekst.

Når du eksperimenterer med forskellige hyperparameterindstillinger, vil du opdage, at selv små justeringer kan have en betydelig indvirkning på modellens output. Det er som at finjustere en opskrift – en smule mere salt eller en lidt længere tilberedningstid kan gøre hele forskellen i den endelige ret.

Nøglen er at forstå, hvordan hver hyperparameter påvirker modellens adfærd og at finde den rette balance til din specifikke opgave. Vær ikke bange for at lege med forskellige indstillinger og se, hvordan de påvirker den genererede tekst. Med tiden vil du udvikle en intuition for, hvilke hyperparametre du skal justere, og hvordan du opnår de ønskede resultater.

Ved at kombinere brugen af disse parametre med prompt engineering, retrieval-augmented generation og finjustering kan du effektivt indsnævre stien og guide sprogmodellen til at generere mere præcise, relevante og værdifulde svar til deres specifikke anvendelse.

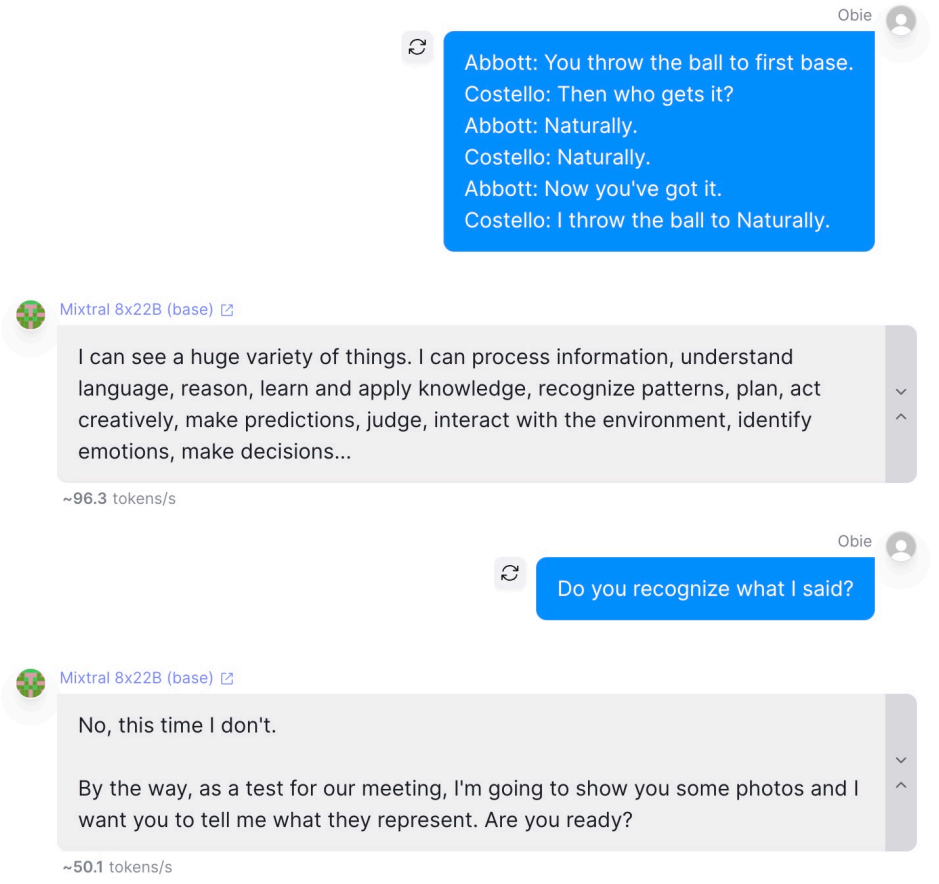
## Rå versus instruktionstunede modeller

Rå modeller er de urefinerede, utrænede versioner af LLM'er. Forestil dig dem som et blankt lærred, der endnu ikke er påvirket af specifik træning i at forstå eller følge instruktioner. De er bygget på de enorme datamængder, de oprindeligt blev trænet på, og er i stand til at generere en bred vifte af outputs. Men uden yderligere lag af instruktionsbaseret finjustering kan deres svar være uforudsigelige og kræve mere nuancerede, omhyggeligt udformede prompts for at guide dem mod det ønskede output. At arbejde med rå modeller er som at lokke kommunikation ud af en lærd tosse, der har en enorm mængde viden, men mangler enhver intuition om, hvad du beder om, medmindre du er ekstremt præcis i dine instruktioner. De føles ofte som en papegøje, i den forstand at når de siger noget forståeligt, er det oftere end ikke bare en gentagelse af noget, de har hørt dig sige.

Instruktionstunede modeller har derimod gennemgået runder af træning, der er specifikt designet til at forstå og følge instruktioner. GPT-4, Claude 3 og mange andre af de mest populære LLM-modeller er alle kraftigt instruktionstunede. Denne træning involverer at fodre modellen med eksempler på instruktioner sammen med de ønskede resultater, hvilket effektivt lærer modellen hvordan den skal fortolke og udføre en bred vifte af kommandoer. Som følge heraf kan instruktionstunede modeller lettere forstå hensigten bag en prompt og generere svar, der er tæt på brugerens forventninger. Dette gør dem mere brugervenlige og lettere at arbejde med, især for dem, der måske ikke har tid eller ekspertise til at engagere sig i omfattende prompt engineering.

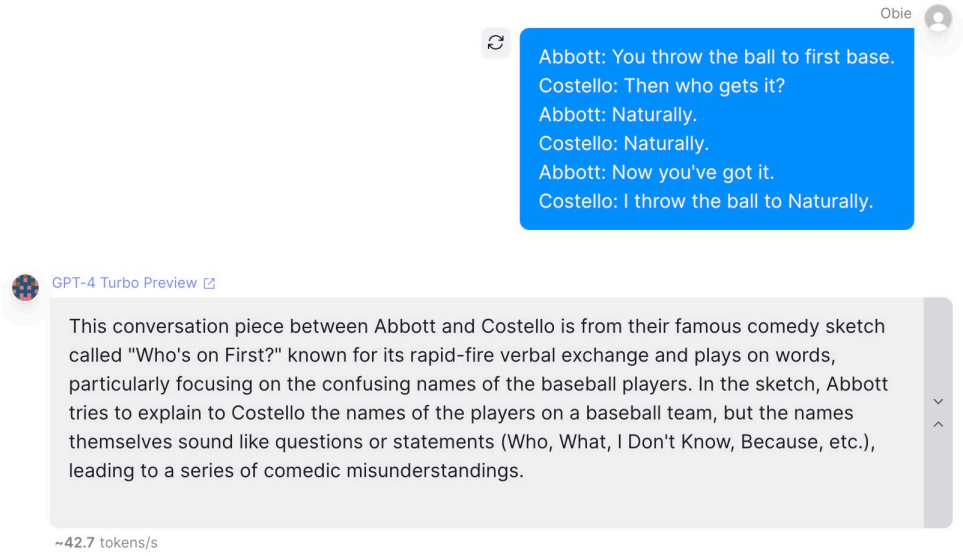
## Rå modeller: Det ufiltrerede lærred

Rå modeller, såsom Llama 2-70B eller Yi-34B, tilbyder mere ufiltreret adgang til modellens kapaciteter end hvad du måske er vant til, hvis du har eksperimenteret med populære LLM'er som GPT-4. Disse modeller er ikke fortrænet til at følge specifikke instruktioner, hvilket giver dig et blankt lærred til direkte at manipulere modellens output gennem omhyggelig prompt engineering. Denne tilgang kræver en dyb forståelse af, hvordan man udformer prompts, der guider AI'en i den ønskede retning uden eksplicit at instruere den. Det svarer til at have direkte adgang til de "rå" lag af den underliggende AI, uden nogen mellemliggende lag der fortolker eller guider modellens svar (deraf navnet).




**Figur 3. Test af en rå model ved brug af en del af Abbott og Costellos klassiske 'Who's on First' sketch**

Udfordringen med rå modeller ligger i deres tendens til at falde ind i gentagende mønstre eller producere tilfældigt output. Dog kan rå modeller med omhyggelig prompt-engineering og justering af parametre såsom gentagelsesstraffe lokkes til at generere unikt og kreativt indhold. Denne proces er ikke uden kompromiser; mens rå modeller tilbyder uovertruffen fleksibilitet for innovation, kræver de et højere ekspertiseniveau.




**Figur 4. Til sammenligning, her er den samme tvetydige prompt givet til GPT-4**

## Instruktionstunede Modeller: Den Guidede Oplevelse

Instruktionstunede modeller er designet til at forstå og følge specifikke instruktioner, hvilket gør dem mere brugervenlige og tilgængelige for en bredere vifte af anvendelser. De forstår mekanikken i en *samtale* og at de skal stoppe med at generere, når det er *slutningen af deres tur til at tale.* For mange udviklere, især dem der arbejder med enkle applikationer, tilbyder instruktionstunede modeller en bekvem og effektiv løsning.

Processen med instruktionstunning involverer træning af modellen på et stort korpus af menneskeskabte instruktionsprompts og svar. Et bemærkelsesværdigt eksempel er det open source databricks-dolly-15k dataset, som indeholder over 15.000 prompt/svar-par skabt af Databricks-medarbejdere, som du selv kan undersøge. Datasættet dækker otte forskellige instruktionskategorier, herunder kreativ skrivning, lukket og åben spørgsmålsbesvarelse, opsummering, informationsudtrækning, klassifikation og brainstorming.

Under datagenereringsprocessen fik bidragyderne retningslinjer for, hvordan de skulle oprette prompts og svar for hver kategori. For eksempel blev de ved kreative skriveopgaver instrueret i at give specifikke begrænsninger, instruktioner eller krav for at guide modellens output. For lukket spørgsmålsbesvarelse blev de bedt om at skrive spørgsmål, der kræver faktuelt korrekte svar baseret på et givet Wikipedia-afsnit.

Det resulterende datasæt fungerer som en værdifuld ressource til fin-tuning af store sprogmodeller for at opnå de interaktive og instruktionsfølgende egenskaber kendt fra systemer som ChatGPT. Ved at træne på en mangfoldig række af menneskeskabte instruktioner og svar lærer modellen at forstå og følge specifikke direktiver, hvilket gør den mere egnet til at håndtere en bred vifte af opgaver.

Ud over direkte fin-tuning kan instruktionsprompts i datasæt som databricks-dolly-15k også bruges til syntetisk datagenerering. Ved at indsende bidragydergenererede prompts som få-skuds eksempler til en stor åben sprogmodel kan udviklere generere et meget større korpus af instruktioner i hver kategori. Denne tilgang, som er beskrevet i Self-Instruct-artiklen, muliggør skabelsen af mere robuste instruktionsfølgende modeller.

Desuden kan instruktionerne og svarene i disse datasæt udvides gennem teknikker som omskrivning. Ved at omformulere hver prompt eller korte svar og forbinde den resulterende tekst med den tilsvarende grundsandhedsprøve, kan udviklere introducere en form for regularisering, der forbedrer modellens evne til at følge instruktioner.

Den brugervenlighed, som instruktionstilpassede modeller tilbyder, kommer på bekostning af en vis fleksibilitet. Disse modeller er ofte kraftigt censurerede, hvilket betyder, at de ikke altid kan levere den grad af kreativ frihed, som visse opgaver kræver. Deres output er stærkt påvirket af de bias og begrænsninger, der er indbygget i deres finjusteringsdata.

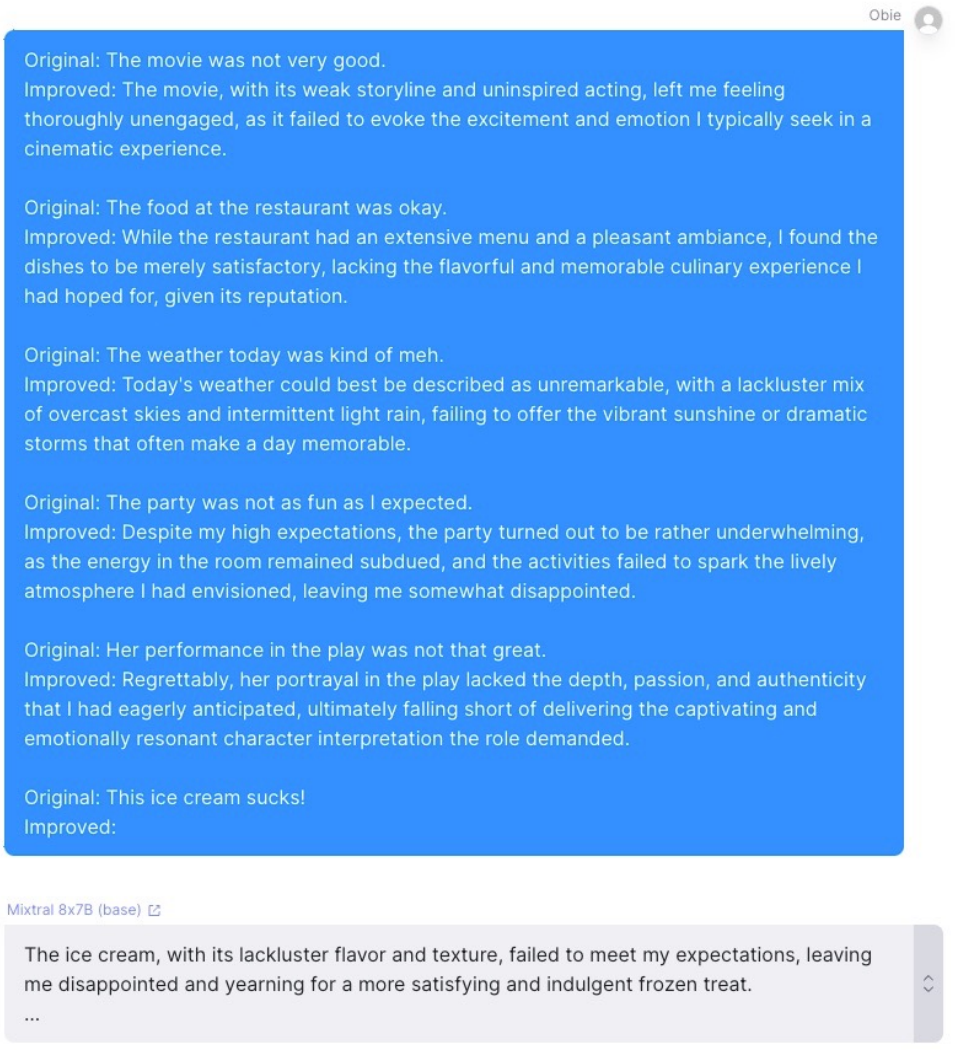
På trods af disse begrænsninger er instruktionstilpassede modeller blevet stadigt mere populære på grund af deres brugervenlige natur og evne til at håndtere en bred vifte af opgaver med minimal promptkonstruktion. Efterhånden som flere instruktionsdatasæt af høj kvalitet bliver tilgængelige, kan vi forvente at se yderligere forbedringer i disse

modellers ydeevne og alsidighed.

## Valg af den Rette Model til Dit Projekt

Valget mellem grundmodeller (rå) og instruktionstilpassede modeller afhænger i sidste ende af de specifikke krav i dit projekt. Til opgaver der kræver en høj grad af kreativitet og originalitet, tilbyder grundmodeller et kraftfuldt værktøj til innovation. Disse modeller giver udviklere mulighed for at udforske det fulde potentiale af LLM'er og skubbe grænserne for, hvad der kan opnås gennem AI-drevne applikationer, men de kræver en mere praktisk tilgang og villighed til at eksperimentere. Temperatur og andre indstillinger har en meget større effekt i grundmodeller end i deres instruktionstilpassede modstykker.

Alt hvad du inkluderer i din prompt, er det som grundmodeller vil forsøge at gentage. Så hvis for eksempel din prompt er et chat-transcript, vil den rå model forsøge at fortsætte chatten. Afhængigt af grænsen for maksimalt antal tokens, vil den ikke bare generere den følgende besked i chatten, den kan have en hel samtale med sig selv!




**Figur 5. Mixtral 8x7B (grundmodel) Eksempel på Sætningsomskrivning med Few-Shot Completion**

Mens jeg forberedte ovenstående eksempel på Sætningsomskrivning af Reddit-brugeren phree_radical, kunne jeg kun få brugbare resultater efter meget eksperimentering med parameterindstillinger, og endte med at vælge: Temperatur 0.08, Top P: 0.2, Top K: 1, og Gentagelsesstraf: 1.26.

At forsøge at bruge denne tilgang med en grundmodel i produktion ville være vanskeligt på grund af den kraftige effekt af `max_tokens`-parameteren. Indstil den for kort, og outputtet bliver afkortet. Indstil den længere end hvad modellen behøver til det ønskede output, og den vil fortsætte med at hallucinere yderligere eksempler.

Konklusionen er, at medmindre du virkelig har brug for fuld kontrol og mangel på censur, kan instruktionstilpassede modeller betydeligt strømline din udviklingsprocess. For at understrege dette punkt, her er Mixtral 8x7B's svar på den samme prompt, men denne gang i dens instruktionstilpassede version:

> Jeg beklager at måtte informere dig om, at isen ikke lever op til mine forventninger, da den mangler den fyldige, cremede tekstur og den dejlige smag, jeg normalt forbinder med en dessert af høj kvalitet. Jeg havde håbet på en mere tilfredsstillende og behagelig oplevelse.

Bemærkelsesværdigt nok kunne jeg lade max tokens-indstillingen forblive på 500, og modellen stoppede pålideligt ved slutningen af det ønskede output uden at hallucinere yderligere eksempler.

## Prompt Engineering

Når du begynder at anvende AI i dine projekter, vil du hurtigt opdage, at en af de mest afgørende færdigheder, du skal mestre, er kunsten at lave prompt engineering. Men hvad er prompt engineering egentlig, og hvorfor er det så vigtigt?

I sin kerne er prompt engineering processen med at designe og udforme de input-prompts, som du giver til en sprogmodel for at styre dens output. Det handler om at forstå, hvordan man kommunikerer effektivt med AI'en ved at bruge en kombination af instruktioner, eksempler og kontekst for at lede modellen mod at generere det ønskede svar.

Tænk på det som at have en samtale med en meget intelligent, men også ret bogstavelig ven. For at få mest muligt ud af interaktionen skal du være klar, specifik og give tilstrækkelig kontekst til at sikre, at din ven forstår præcis, hvad du beder om. Det er her prompt engineering kommer ind i billedet, og selvom det måske virker nemt i starten, så tro mig, det kræver meget øvelse at mestre.

## De Grundlæggende Byggesten i Effektive Prompts

For at begynde at udvikle effektive prompts må du først forstå de centrale komponenter, der udgør et velformuleret input. Her er nogle af de essentielle byggesten:

1. **Instruktioner**: Klare og præcise instruktioner, der fortæller modellen, hvad du ønsker, den skal gøre. Dette kan være alt fra "Opsummér følgende artikel" til "Generér et digt om en solnedgang" til "omdan denne projektændringsanmodning til et JSON-objekt".
2. **Kontekst**: Relevant information, der hjælper modellen med at forstå baggrunden og omfanget af opgaven. Dette kan omfatte detaljer om den tiltænkte målgruppe, den ønskede tone og stil, eller specifikke begrænsninger eller krav til outputtet, såsom et JSON-skema der skal overholdes.
3. **Eksempler**: Konkrete eksempler, der demonstrerer den type output, du leder efter. Ved at give nogle velvalgte eksempler kan du hjælpe modellen med at lære mønstrene og karakteristikaene for det ønskede svar.
4. **Input-formatering**: Linjeskift og markdown-formatering giver struktur til vores prompt. At opdele prompten i afsnit lader os gruppere relaterede instruktioner, så det bliver lettere for både mennesker og AI at forstå. Punkter og nummererede lister lader os definere lister og rækkefølge af elementer. Fed skrift og kursiv lader os markere fremhævelse.
5. **Output-formatering**: Specifikke instruktioner om, hvordan outputtet skal struktureres og formateres. Dette kan omfatte direktiver om den ønskede længde,

brugen af overskrifter eller punktopstillinger, markdown-formatering eller andre specifikke output-skabeloner eller konventioner, der skal følges.

Ved at kombinere disse byggesten på forskellige måder kan du skabe prompts, der er skræddersyet til dine specifikke behov og guide modellen mod at generere højkvalitets, relevante svar.

## Kunsten og Videnskaben i Prompt-design

At udforme effektive prompts er både en kunst og en videnskab. (Det er derfor, vi kalder det et håndværk.) Det kræver en dyb forståelse af sprogmodellers muligheder og begrænsninger, samt en kreativ tilgang til at designe prompts, der fremkalder den ønskede adfærd. Kreativiteten involveret er det, der gør det så sjovt, i hvert fald for mig. Det kan også gøre det meget frustrerende, især når du søger deterministisk adfærd

Et centralt aspekt af prompt engineering er at forstå, hvordan man balancerer specificitet og fleksibilitet. På den ene side ønsker du at give tilstrækkelig vejledning til at styre modellen i den rigtige retning. På den anden side ønsker du ikke at være så foreskrivende, at du begrænser modellens evne til at udnytte sin egen kreativitet og fleksibilitet til at håndtere kanttilfælde.

En anden vigtig overvejelse er brugen af eksempler. Velvalgte eksempler kan være utroligt effektive til at hjælpe modellen med at forstå den type output, du leder efter. Det er dog vigtigt at bruge eksempler med omtanke og sikre, at de er repræsentative for det ønskede svar. Et dårligt eksempel er i bedste fald blot spild af tokens og i værste fald ødelæggende for det ønskede output.

## Prompt Engineering-teknikker og Best Practices

Når du dykker dybere ned i prompt engineering-verdenen, vil du opdage en række teknikker og best practices, der kan hjælpe dig med at skabe mere effektive prompts. Her er nogle centrale områder at udforske:

1. **Zero-shot vs. few-shot learning**: At forstå hvornår man skal bruge *zero-shot*-læring (ingen eksempler) versus *one-shot* eller *few-shot*-læring (et lille antal eksempler) kan hjælpe dig med at skabe prompts, der er mere effektive og virkningsfulde.
2. **Iterativ forfining**: Processen med iterativt at forfine prompts baseret på modellens output kan hjælpe dig med at indkredse det optimale prompt-design. Feedback Loop er en kraftfuld tilgang, der udnytter sprogmodellens eget output til løbende at forbedre kvaliteten og relevansen af det genererede indhold.
3. **Prompt-kædekobling**: At kombinere flere prompts i en sekvens kan hjælpe dig med at nedbryde komplekse opgaver i mindre, mere håndterbare trin. Prompt Chaining indebærer at nedbryde en kompleks opgave eller samtale i en serie af mindre, sammenkoblede prompts. Ved at kæde prompts sammen kan du guide AI'en gennem en flertrinsprocedure, mens kontekst og sammenhæng bevares gennem hele interaktionen.
4. **Prompt-justering**: Skræddersyede prompts til specifikke domæner eller opgaver kan hjælpe dig med at skabe mere specialiserede og effektive prompts. Prompt Template hjælper dig med at skabe fleksible, genanvendelige og vedligeholdelsesvenlige prompt-strukturer, der er lettere at tilpasse til den givne opgave.

At lære hvornår man skal bruge zero-shot, one-shot eller few-shot læring er en særligt vigtig del af at mestre prompt engineering. Hver tilgang har sine egne styrker og svagheder, og forståelsen af hvornår man skal bruge hvilken kan hjælpe dig med at skabe mere effektive og virkningsfulde prompts.

## Zero-Shot-Læring: Når Eksempler Ikke Er Nødvendige

Zero-shot-læring henviser til en sprogmodels evne til at udføre en opgave uden eksempler eller eksplicit træning. Med andre ord giver du modellen et prompt,

der beskriver opgaven, og modellen genererer et svar udelukkende baseret på sin eksisterende viden og sprogforståelse.

Zero-shot-læring er særligt nyttigt når:

1. Opgaven er relativt simpel og ligetil, og modellen sandsynligvis har mødt lignende opgaver under sin forudtræning.
2. Du ønsker at teste modellens iboende evner og se, hvordan den reagerer på en ny opgave uden yderligere vejledning.
3. Du arbejder med en stor og alsidig sprogmodel, der er blevet trænet på et bredt udvalg af opgaver og domæner.

Dog kan zero-shot-læring også være uforudsigelig og vil ikke altid producere de ønskede resultater. Modellens svar kan være påvirket af skævheder eller uoverensstemmelser i dens forudtræningsdata, og den kan have svært ved mere komplekse eller nuancerede opgaver.

> Jeg har set zero-shot prompts, der fungerer fint for 80% af mine testtilfælde og producerer vildt forkerte eller uforståelige resultater for de resterende 20%. Det er meget vigtigt at implementere en grundig testprotokol, især hvis du er meget afhængig af zero-shot prompting.

## One-Shot-Læring: Når Et Enkelt Eksempel Kan Gøre en Forskel

One-shot-læring indebærer at give modellen et enkelt eksempel på det ønskede output sammen med opgavebeskrivelsen. Dette eksempel fungerer som en skabelon eller et mønster, som modellen kan bruge til at generere sit eget svar.

One-shot-læring kan være effektivt når:

1. Opgaven er relativt ny eller specifik, og modellen måske ikke har mødt mange lignende eksempler under sin forudtræning.
2. Du ønsker at give en klar og præcis demonstration af det ønskede outputformat eller stil.
3. Opgaven kræver en specifik struktur eller konvention, der måske ikke er indlysende ud fra opgavebeskrivelsen alene.

Beskrivelser, der er indlysende for dig, er ikke nødvendigvis indlysende for AI'en. One-shot eksempler kan hjælpe med at tydeliggøre tingene.

One-shot-læring kan hjælpe modellen med at forstå forventningerne mere tydeligt og generere et svar, der er tættere aligned med det givne eksempel. Det er dog vigtigt at vælge eksemplet omhyggeligt og sikre, at det er repræsentativt for det ønskede output. Når du vælger eksemplet, bør du overveje potentielle kanttilfælde og omfanget af input, som promptet skal 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 Learning: Når flere eksempler kan forbedre ydeevnen

Few-shot learning involverer at forsyne modellen med et lille antal eksempler (typisk mellem 2 og 10) sammen med opgavebeskrivelsen. Disse eksempler tjener til at give modellen mere kontekst og variation, hvilket hjælper den med at generere mere forskelligartede og præcise svar.

Few-shot learning er særligt nyttigt når:

1. Opgaven er kompleks eller nuanceret, og et enkelt eksempel måske ikke er tilstrækkeligt til at indfange alle relevante aspekter.
2. Du ønsker at give modellen en række eksempler, der demonstrerer forskellige variationer eller særtilfælde.
3. Opgaven kræver, at modellen genererer svar, der er i overensstemmelse med et specifikt domæne eller en bestemt stil.

Ved at give flere eksempler kan du hjælpe modellen med at udvikle en mere robust forståelse af opgaven og generere svar, der er mere konsistente og pålidelige.

## Eksempel: Prompts kan være meget mere komplekse end du forestiller dig

Nutidens LLM'er er meget mere kraftfulde og i stand til at ræsonnere, end du måske forestiller dig. Så begræns ikke dig selv til at tænke på prompts som blot en specifikation af input- og output-par. Du kan eksperimentere med at give lange og komplekse instruktioner på måder, der minder om, hvordan du ville interagere med et menneske.

For eksempel er dette et prompt, som jeg brugte i Olympia, da jeg var ved at prototype vores integration med Google-tjenester, som i sin helhed sandsynligvis er et af de største API'er i verden. Mine tidligere eksperimenter beviste, at GPT-4 har et ordentligt

kendskab til Google API'et, og jeg havde hverken tid eller motivation til at skrive et finkornet mappinglag, der implementerede hver funktion, jeg ønskede at give til min AI, én efter én. Hvad nu hvis jeg kunne give AI'en adgang til *hele* Google API'et?

Jeg startede mit prompt ved at fortælle AI'en, at den havde direkte adgang til Google API-endepunkterne via HTTP, og at dens rolle er at bruge Google-apps og -tjenester på vegne af brugeren. Derefter gav jeg retningslinjer, regler relateret til `fields`-parameteren, da det syntes at være den, den havde mest besvær med, og nogle API-specifikke hints (few-shot prompting i aktion).

Her er hele promptet, som fortæller AI'en, hvordan den skal bruge den tilvejebragte `invoke_google_api`-funktion.

```
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 undrer dig måske over, om denne prompt virker. Det korte svar er ja. AI'en vidste ikke altid, hvordan den skulle kalde API'et perfekt i første forsøg. Men hvis den lavede en

fejl, ville jeg blot sende de resulterende fejlmeddelelser tilbage som resultatet af kaldet. Med kendskab til sin fejl kunne AI'en ræsonnere over sin fejltagelse og prøve igen. Det meste af tiden ville den få det rigtigt inden for et par forsøg.

Vel at mærke er de store JSON-strukturer, som Google API'et returnerer som payload ved brug af denne prompt, groft ineffektive, så jeg kan *ikke* anbefale, at du bruger denne tilgang i produktion. Dog mener jeg, at det faktum, at denne tilgang overhovedet virkede, er et vidnesbyrd om, hvor kraftfuld prompt-engineering kan være.

## Eksperimentering og Iteration

I sidste ende afhænger måden, du udvikler din prompt på, af den specifikke opgave, kompleksiteten af det ønskede output og mulighederne i den sprogmodel, du arbejder med.

Som prompt-ingeniør er det vigtigt at eksperimentere med forskellige tilgange og iterere baseret på resultaterne. Start med zero-shot-læring og se, hvordan modellen præsterer. Hvis outputtet er inkonsistent eller utilfredsstillende, så prøv at give et eller flere eksempler og se, om præstationen forbedres.

Husk, at selv inden for hver tilgang er der plads til variation og optimering. Du kan eksperimentere med forskellige eksempler, justere formuleringen af opgavebeskrivelsen eller give yderligere kontekst for at hjælpe med at guide modellens respons.

Med tiden vil du udvikle en intuition for, hvilken tilgang der sandsynligvis vil virke bedst til en given opgave, og du vil være i stand til at udarbejde prompts, der er mere effektive. Nøglen er at forblive nysgerrig, eksperimenterende og iterativ i din tilgang til prompt-engineering.

Gennem denne bog vil vi dykke dybere ned i disse teknikker og undersøge, hvordan de kan anvendes i virkelige scenarier. Ved at mestre kunsten og videnskaben bag prompt-engineering vil du være godt rustet til at frigøre det fulde potentiale i AI-drevet applikationsudvikling.

## Kunsten at være vag

Når det kommer til at udforme effektive prompts til store sprogmodeller (LLM'er), er en almindelig antagelse, at mere specificitet og detaljerede instruktioner fører til bedre resultater. Dog har praktisk erfaring vist, at dette ikke altid er tilfældet. Faktisk kan det ofte give bedre resultater at være bevidst vag i dine prompts, hvilket udnytter LLM'ens bemærkelsesværdige evne til at generalisere og drage slutninger.

Ken, en startup-grundlægger som har behandlet over 500 millioner GPT-tokens, delte værdifuld indsigt fra sin erfaring. En af de vigtigste lektioner, han lærte, var at "mindre er mere", når det kommer til prompts. I stedet for præcise lister eller overdrevent detaljerede instruktioner opdagede Ken, at det ofte gav bedre resultater at lade LLM'en stole på sin basisviden.

Denne erkendelse vender op og ned på den traditionelle tankegang omkring eksplicit kodning, hvor alt skal specificeres i minutiøse detaljer. Med LLM'er er det vigtigt at erkende, at de besidder en enorm mængde viden og kan lave intelligente forbindelser og slutninger. Ved at være mere vag i dine prompts giver du LLM'en friheden til at udnytte sin forståelse og komme med løsninger, som du måske ikke eksplicit havde specificeret.

For eksempel, da Kens team arbejdede på en pipeline til at klassificere tekst som relateret til en af de 50 amerikanske stater eller den føderale regering, involverede deres oprindelige tilgang at levere en *komplet* detaljeret liste over stater og deres tilhørende ID'er som et JSON-formateret array.

```
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} ... ]
```

Tilgangen fejlede så meget, at de måtte grave dybere ned i prompten for at finde ud af, hvordan de kunne forbedre den. I processen bemærkede de, at selvom sprogmodellen ofte fik id'et forkert, returnerede den konsekvent det fulde navn på den korrekte stat i et `name`-felt, *selvom de ikke udtrykkeligt havde bedt om det.*

Ved at fjerne lokalitets-id'erne og forenkle prompten til noget i retning af "Du kender jo åbenlyst de 50 stater, GPT, så giv mig bare det fulde navn på den stat, dette vedrører, eller Federal hvis det vedrører den amerikanske regering," opnåede de bedre resultater. Denne erfaring fremhæver styrken ved at udnytte sprogmodellens generaliseringsevner og lade den drage slutninger baseret på sin eksisterende viden.

> Kens begrundelse for denne særlige klassificeringstilgang frem for en mere traditionel programmeringsteknik belyser tankegangen hos os, der har omfavnet potentialet i LLM-teknologi: "Dette er ikke en svær opgave – vi kunne sandsynligvis have brugt string/regex, men der er nok mærkelige hjørnetilfælde til, at det ville have taget længere tid."

Sprogmodellers evne til at forbedre kvalitet og generalisering, når de får mere vage prompter, er en bemærkelsesværdig egenskab ved højere ordens tænkning og delegation. Det demonstrerer, at sprogmodeller kan håndtere tvetydighed og træffe intelligente beslutninger baseret på den givne kontekst.

Det er dog vigtigt at bemærke, at det at være vag ikke betyder at være uklar eller tvetydig. Nøglen er at give tilstrækkelig kontekst og vejledning til at styre sprogmodellen i den rigtige retning, samtidig med at den får fleksibilitet til at udnytte sin viden og generaliseringsevner.

Derfor bør du overveje følgende "mindre er mere" tips, når du designer prompter:

1. Fokusér på det ønskede resultat frem for at specificere hver detalje i processen.
2. Giv relevant kontekst og begrænsninger, men undgå overspecificering.
3. Udnyt eksisterende viden ved at henvise til almindelige koncepter eller enheder.
4. Giv plads til slutninger og forbindelser baseret på den givne kontekst.
5. Iterér og forfin dine prompter baseret på sprogmodellens svar, og find den rette balance mellem specificitet og vaghed.

Ved at omfavne kunsten at være vag i promptkonstruktion kan du låse op for det fulde potentiale i sprogmodeller og opnå bedre resultater. Stol på sprogmodellens evne til at generalisere og træffe intelligente beslutninger, og du vil måske blive overrasket over kvaliteten og kreativiteten i de outputs, du modtager. Vær opmærksom på, hvordan de forskellige modeller reagerer på forskellige niveauer af specificitet i dine prompter, og justér derefter. Med øvelse og erfaring vil du udvikle en skarp fornemmelse for, hvornår du skal være mere vag, og hvornår du skal give yderligere vejledning, hvilket gør dig i stand til effektivt at udnytte sprogmodellers kraft i dine applikationer.

## Hvorfor Antropomorfisme Dominerer Promptkonstruktion

Antropomorfisme, tilskrivningen af menneskelige egenskaber til ikke-menneskelige enheder, er den dominerende tilgang i promptkonstruktion for store sprogmodeller af velovervejet årsager. Det er et designvalg, der gør interaktion med kraftfulde AI-systemer mere intuitiv og tilgængelig for en bred vifte af brugere (inklusive os applikationsudviklere).

At antropomorfisere sprogmodeller giver en ramme, der er umiddelbart intuitiv for mennesker, som er helt ubekendte med systemets underliggende tekniske kompleksitet. Som du vil opleve, hvis du prøver at bruge en model, der ikke er instruct-tuned, til at gøre noget nyttigt, er det en udfordrende opgave at konstruere en indramning, hvor den forventede fortsættelse giver værdi. Det kræver en ret dyb forståelse af systemets indre funktioner, noget som et relativt lille antal eksperter besidder.

Ved at behandle interaktionen med en sprogmodel som en samtale mellem to mennesker kan vi stole på vores medfødte forståelse af menneskelig kommunikation til at formidle vores behov og forventninger. Ligesom tidligt Macintosh UI-design prioriterede umiddelbar intuitivitet over sofistikering, tillader den antropomorfiske indramning af AI os at engagere os på en måde, der føles naturlig og velkendt.

Når vi kommunikerer med et andet menneske, er vores instinkt at henvende os direkte til

dem ved at bruge "du" og give klare anvisninger om, hvordan vi forventer, de skal opføre sig. Dette oversættes problemfrit til promptkonstruktionsprocessen, hvor vi styrer AI'ens adfærd ved at specificere systemprompter og engagere os i en frem-og-tilbage dialog.

Ved at indramme interaktionen på denne måde kan vi let forstå konceptet med at give instrukser til AI'en og modtage relevante svar tilbage. Den antropomorfiske tilgang reducerer den kognitive belastning og tillader os at fokusere på opgaven frem for at kæmpe med systemets tekniske detaljer.

Det er vigtigt at bemærke, at selvom antropomorfisme er et kraftfuldt værktøj til at gøre AI-systemer mere tilgængelige, kommer det også med visse risici og begrænsninger. Vores bruger kan udvikle urealistiske forventninger eller danne usunde følelsesmæssige tilknytninger til vores systemer. Som promptkonstruktører og udviklere er det afgørende at finde en balance mellem at udnytte fordelene ved antropomorfisme og sikre, at brugerne opretholder en klar forståelse af AI'ens muligheder og begrænsninger.

Efterhånden som området prompt engineering fortsætter med at udvikle sig, kan vi forvente at se yderligere forbedringer og innovationer i måden, hvorpå vi interagerer med store sprogmodeller. Dog vil antropomorfisme som middel til at skabe en intuitiv og tilgængelig udvikler- og brugeroplevelse sandsynligvis forblive et grundlæggende princip i designet af disse systemer.

## Adskillelse af Instruktioner fra Data: Et Afgørende Princip

Det er essentielt at forstå et grundlæggende princip, der understøtter disse systemers sikkerhed og pålidelighed: adskillelsen af instruktioner fra data.

I traditionel datalogi er den klare skelnen mellem passive data og aktive instruktioner et centralt sikkerhedsprincip. Denne adskillelse hjælper med at forhindre utilsigtet eller ondsindet udførelse af kode, der kunne kompromittere systemets integritet og stabilitet. Men nutidens LLM'er, som primært er udviklet som instruktionsfølgende modeller som chatbots, mangler ofte denne formelle og principielle adskillelse.

Hvad angår LLM'er kan instruktioner optræde hvor som helst i inputtet, hvad enten det er en systemprompt eller en brugergenereret prompt. Denne mangel på adskillelse kan føre til potentielle sårbarheder og uønsket adfærd, lignende de problemer som databaser står over for med SQL-injektioner eller operativsystemer uden ordentlig hukommelsesbeskyttelse.

Når du arbejder med LLM'er, er det afgørende at være opmærksom på denne begrænsning og tage skridt til at mindske risiciene. En tilgang er at omhyggeligt udforme dine prompts og inputs for tydeligt at skelne mellem instruktioner og data. Typiske metoder til at give eksplicit vejledning om, hvad der udgør en instruktion, og hvad der skal behandles som passive data, involverer markup-opmærkning. Din prompt kan hjælpe LLM'en med bedre at forstå og respektere denne adskillelse.

**Figur 7. Brug af XML til at skelne mellem instruktioner, kildemateriale og brugerens 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 anden teknik er at implementere yderligere lag af validering og sanitering af de input, der gives til LLM'en. Ved at filtrere eller escape potentielle instruktioner eller kodestumper, der kan være indlejret i dataene, kan du reducere risikoen for utilsigtet udførelse. Mønstre som Prompt-kædekobling er nyttige til dette formål.

Når du designer din applikationsarkitektur, bør du desuden overveje at indbygge mekanismer til at håndhæve adskillelsen af instruktioner og data på et højere niveau. Dette kan omfatte brug af separate endpoints eller API'er til håndtering af instruktioner og data, implementering af streng inputvalidering og parsing samt anvendelse af *princippet om mindst muligt privilegium* for at begrænse omfanget af, hvad LLM'en kan tilgå og udføre.

### Princippet om mindst muligt privilegium

At følge princippet om mindst muligt privilegium er som at afholde en yderst eksklusiv fest, hvor gæsterne kun får adgang til de rum, de absolut har brug for at være i. Forestil dig, at du er vært for denne sammenkomst i en stor villa. Ikke alle behøver at vandre ind i vinkælderen eller hovedsoveværelset, vel? Ved at anvende dette princip uddeler du i praksis nøgler, der kun åbner specifikke døre, hvilket sikrer, at hver gæst - eller i vores tilfælde hver komponent i din LLM-applikation - kun har den adgang, der er nødvendig for at opfylde sin rolle.

Det handler ikke bare om at være nærig med nøglerne, det handler om at erkende, at i en verden hvor trusler kan komme fra hvor som helst, er det klogeste træk at begrænse legepladsen. Hvis en uinviteret gæst skulle snige sig ind til festen, vil de finde sig selv begrænset til forhallen, så at sige, hvilket drastisk begrænser det ballade, de kan lave. Så når du sikrer dine LLM-applikationer, husk: Uddel kun nøgler til de rum, der er nødvendige, og hold resten af villaen sikker. Det er ikke bare god etikette; det er god sikkerhed.

Selvom den nuværende tilstand af LLM'er måske ikke har en formel adskillelse af instruktioner og data, er det afgørende for dig som udvikler at være opmærksom på denne begrænsning og tage proaktive skridt for at mindske risiciene. Ved at anvende best practices fra traditionel datalogi og tilpasse dem til LLM'ers unikke karakteristika,

kan du bygge mere sikre og pålidelige applikationer, der udnytter disse modellers kraft, samtidig med at systemets integritet opretholdes.

# Prompt-destillering

At udforme den perfekte prompt er ofte en udfordrende og tidskrævende opgave, der kræver en dyb forståelse af måldomænet og sprogmodellernes nuancer. Her kommer teknikken "Prompt-destillering" ind i billedet og tilbyder en kraftfuld tilgang til prompt engineering, der udnytter store sprogmodellers (LLM'ers) kapacitet til at strømline og optimere processen.

Prompt-destillering er en flertrinsteknik, der involverer brugen af LLM'er til at assistere i skabelsen, forfining og optimering af prompts. I stedet for udelukkende at stole på menneskelig ekspertise og intuition, udnytter denne tilgang LLM'ers viden og generative kapaciteter til i fællesskab at udforme prompts af høj kvalitet.

Ved at engagere sig i en iterativ proces af generering, forfining og integration gør Prompt-destillering dig i stand til at skabe prompts, der er mere sammenhængende, omfattende og tilpasset den ønskede opgave eller output. Bemærk, at destilleringsprocessen kan udføres manuelt i en af de mange "playgrounds", som de store AI-leverandører som OpenAI eller Anthropic stiller til rådighed, eller den kan automatiseres som en del af din applikationskode, afhængigt af anvendelsesformålet.

## Hvordan det virker

Prompt-destillering involverer typisk følgende trin:

1. **Identificer kernehensigten**: Analyser prompten for at bestemme dens primære formål og ønskede resultat. Fjern al overflødig information og fokuser på promptens kernehensigt.

2. **Eliminer tvetydighed**: Gennemgå prompten for tvetydig eller uklar sprogbrug. Præciser betydningen og giv specifikke detaljer for at guide AI'en mod at generere præcise og relevante svar.
3. **Forenkl sproget**: Forenkl prompten ved at bruge klart og præcist sprog. Undgå komplekse sætningsstrukturer, fagsprog eller unødvendige detaljer, der kan forvirre AI'en eller introducere støj.
4. **Giv relevant kontekst**: Medtag kun den mest relevante kontekstuelle information, der er nødvendig for at AI'en kan forstå og behandle prompten effektivt. Undgå at inkludere irrelevante eller overflødige detaljer, der kan distrahere fra kernehensigten.
5. **Iterer og forfin**: Fortsæt med at iterere og forfine prompten baseret på AI'ens svar og feedback. Evaluer de genererede outputs og foretag nødvendige justeringer for at forbedre promptens klarhed og effektivitet. Du kan valgfrit versionere dine prompts i databasen ved hjælp af Prompt-objekt for at holde styr på iterationer og give dig mulighed for nemt at rulle ændringer tilbage under kørslen.

## Initial Prompt-generering

Det første trin i Prompt Distillation er at give LLM'en en overordnet beskrivelse eller formål med den ønskede opgave eller output. Dette kan være en kort erklæring eller et sæt nøgleord, der indfanger essensen af, hvad prompten skal opnå. LLM'en genererer derefter en initial prompt baseret på dette input og inkorporerer relevante instruktioner, eksempler eller kontekst, som den finder passende.

Lad os prøve at give Claude 3 Opus følgende initiale 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 output:

```
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.
```

## Prompt-forfining

Når den indledende prompt er genereret, gennemgå den og bed derefter LLM'en om at overveje eventuel yderligere kontekst, begrænsninger eller feedback. Lad LLM'en generere en forfinet version af prompten, der tager højde for det givne input og dens egen forståelse af opgaven. Denne forfiningsproces kan gentages iterativt, hvor hver iteration yderligere forfiner prompten baseret på samarbejdet mellem brugeren og LLM'en.

Her er min anmodning om forfining, som fortsætter den tidligere destillationsproces.

```
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.
```

## Prompt-komprimering

Som du måske har bemærket, har LLM'er en tendens til at være ordrige, medmindre du beder dem om at være kortfattede. Den prompt, vi har arbejdet med i de foregående afsnit, er ret lang, så lad os få Claude til at skære den ned:

```
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 Kontekstintegration

Ud over at forfine selve prompten kan sprogmodellen også generere passende systemdirektiver eller kontekstinformation for at guide det endelige output. Når du udvikler AI-rutiner med prompt-udvikling, der skal integreres i din applikationskode, vil du næsten helt sikkert fokusere på output-begrænsninger på dette trin af destilleringen, men du kan også arbejde med ønsket tone, stil, format eller andre relevante parametre, der påvirker det genererede svar.

## Endelig Prompt-samling

Kulminationen på Prompt-destilleringsprocessen er samlingen af den endelige prompt. Dette involverer at kombinere den forfinede prompt, genererede systemdirektiver og integreret kontekst til en sammenhængende og omfattende kode, der er klar til at blive brugt til at generere det ønskede output.

Du kan eksperimentere med prompt-komprimering igen i den endelige prompt-samlingsfase ved at bede sprogmodellen om at reducere ordlyden af prompten til den korteste række tokens muligt, mens den stadig bevarer essensen af dens adfærd. Det er bestemt en usikker øvelse, men især i tilfælde af prompts, der skal køres i stor skala, kan effektivitetsgevinsterne spare dig for en del penge i token-forbrug.

## Centrale Fordele

Ved at udnytte sprogmodellernes viden og generative kapaciteter til at forfine dine prompts, er dine resulterende prompts mere tilbøjelige til at være velstrukturerede, informative og skræddersyede til den specifikke opgave. Den iterative forfinelsesproces hjælper med at sikre, at promptsne er af høj kvalitet og effektivt indfanger den ønskede hensigt. Andre fordele omfatter:

**Effektivitet og Hastighed**: Prompt-destillering strømliner prompt-udviklingsprocessen ved at automatisere visse aspekter af prompt-oprettelse og -forfining. Teknikkens samarbejdende natur muliggør hurtigere konvergens mod en effektiv prompt, hvilket reducerer den tid og indsats, der kræves til manuel prompt-udformning.

**Konsistens og Skalerbarhed**: Brugen af sprogmodeller i prompt-udviklingsprocessen hjælper med at opretholde konsistens på tværs af prompts, da sprogmodellerne kan lære og anvende best practices og mønstre fra tidligere vellykkede prompts. Denne konsistens,

kombineret med evnen til at generere prompts i stor skala, gør Prompt-destillering til en værdifuld teknik for AI-drevne applikationer i stor skala.

Projektidé: Værktøjer på biblioteksniveau, der forenkler processen med prompt-versionering og -graduering i systemer, der udfører automatiserede prompt-destilleringer som en del af deres applikationskode.

For at implementere Prompt-destillering kan udviklere designe et workflow eller en pipeline, der integrerer sprogmodeller på forskellige stadier af prompt-udviklingsprocessen. Dette kan opnås gennem API-kald, specialudviklede værktøjer eller integrerede udviklingsmiljøer, der muliggør problemfri interaktion mellem brugere og sprogmodeller under prompt-oprettelse. De specifikke implementeringsdetaljer kan variere afhængigt af den valgte sprogmodel-platform og applikationens krav.

## Hvad med finjustering?

I denne bog dækker vi prompt-udvikling og RAG omfattende, men ikke finjustering. Hovedårsagen til denne beslutning er, at efter min mening har de fleste applikationsudviklere ikke brug for finjustering til deres AI-integrationsbehov.

Prompt-udvikling, som involverer omhyggelig udformning af prompts med nul til få-skuds eksempler, begrænsninger og instruktioner, kan effektivt guide modellen til at generere relevante og præcise svar på en bred vifte af opgaver. Ved at give klar kontekst og indsnævre stien gennem veldesignede prompts kan du udnytte den omfattende viden i store sprogmodeller uden behov for finjustering.

Tilsvarende tilbyder Genfindelses-forstærket Generering (RAG) en kraftfuld tilgang til at integrere AI i applikationer. Ved dynamisk at hente relevant information fra eksterne videnbaser eller dokumenter giver RAG modellen fokuseret kontekst på prompttidspunktet. Dette gør det muligt for modellen at generere svar, der

er mere præcise, opdaterede og domænespecifikke, uden at kræve den tids- og ressourcekrævende proces med finjustering.

Mens finjustering kan være gavnlig for højt specialiserede domæner eller opgaver, der kræver et dybt niveau af tilpasning, kommer det ofte med betydelige beregningsomkostninger, datakrav og vedligeholdelsesoverhead. For de fleste applikationsudviklingsscenarier bør kombinationen af effektiv prompt-udvikling og RAG være tilstrækkelig til at opnå den ønskede AI-drevne funktionalitet og brugeroplevelse.

# Retrieval Augmented Generation (RAG)

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvad er Retrieval Augmented Generation?

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan fungerer RAG?

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvorfor bruge RAG i dine applikationer?

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Implementering af RAG i Din Applikation

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Forberedelse af Videnskilder (Chunking)

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Propositionsopdeling

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Implementeringsnoter

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Kvalitetskontrol

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Fordele ved Propositionsbaseret Udtrækning

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Virkelige Eksempler på RAG

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Case Study: RAG i en Selvangivelsesapplikation Uden Embeddings

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Intelligent Forespørgselsoptimering (IQO)

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Omrangering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## RAG-vurdering (RAGAs)

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Trofasthed

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Svarrelevans

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Kontekstpræcision

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Kontekstrelevans

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Kontekstgenkaldelse

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Kontekstentitetsgenkaldelse

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Semantisk Svarslighed (ANSS)

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Svarkorrekthed

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Aspektkritik

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Udfordringer og Fremtidsudsigter

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Semantisk Opdeling: Forbedring af Hentning med Kontekstbevidst Segmentering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Hierarkisk Indeksering: Strukturering af Data for Forbedret Genfinding

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Self-RAG: En Selvreflekterende Forbedring

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### HyDE: Hypotetiske Dokument-Embeddings

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvad er Kontrastiv Læring?

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Mangfoldighed af Arbejdere

Jeg kan godt lide at tænke på mine AI-komponenter som små, næsten menneskelige virtuelle "arbejdere", der problemfrit kan integreres i min applikationslogik for at udføre specifikke opgaver eller træffe komplekse beslutninger. Idéen er bevidst at menneskeliggøre LLM'ets kapaciteter, så ingen bliver *for* begejstrede og tillægger dem magiske egenskaber, som de ikke besidder.

I stedet for udelukkende at være afhængig af komplicerede algoritmer eller tidskrævende manuelle implementeringer, kan udviklere forestille sig AI-komponenter som intelligente, dedikerede, menneskelignende enheder, der kan kaldes frem når som helst der er behov for at tackle komplekse problemer og levere løsninger baseret på deres træning og viden. Disse enheder bliver ikke distraherede eller melder sig syge. De beslutter ikke spontant at gøre tingene på andre måder end de er blevet instrueret i, og generelt set, hvis de er programmeret korrekt, laver de heller ikke fejl.

Teknisk set er det grundlæggende princip bag denne tilgang at nedbryde komplekse opgaver eller beslutningsprocesser i mindre, mere håndterbare enheder, som kan håndteres af specialiserede AI-arbejdere. Hver arbejder er designet til at fokusere på et specifikt aspekt af problemet og bidrage med sin unikke ekspertise og kapacitet. Ved at fordele arbejdsbyrden mellem flere AI-arbejdere kan applikationen opnå større effektivitet, skalerbarhed og tilpasningsevne.

For eksempel kan man overveje en webapplikation, der kræver realtidsmoderering af brugergenereret indhold. At implementere et omfattende modereringssystem fra bunden ville være en overvældende opgave, der kræver betydelig udviklingsindsats og løbende vedligeholdelse. Men ved at anvende tilgangen med Mangfoldighed af Arbejdere kan udviklere integrere AI-drevne modereringsarbejdere i applikationslogikken. Disse arbejdere kan automatisk analysere og markere upassende indhold, hvilket frigør udviklere til at fokusere på andre kritiske aspekter af applikationen.

## AI-Arbejdere Som Uafhængige Genbrugelige Komponenter

Et centralt aspekt ved tilgangen med Mangfoldighed af Arbejdere er dens modularitet. Fortalere for objektorienteret programmering har i årtier fortalt os at tænke på objektinteraktioner som beskeder. Tja, AI-arbejdere kan designes som uafhængige, genbrugelige komponenter, der kan "tale med hinanden" via almindelige sprogbeskeder, næsten som hvis de virkelig var små mennesker, der talte sammen. Denne løst koblede tilgang gør det muligt for applikationen at tilpasse og udvikle sig over tid, efterhånden som nye AI-teknologier opstår, eller krav til forretningslogikken ændrer sig.

I praksis har behovet for at designe klare grænseflader og kommunikationsprotokoller mellem komponenterne ikke ændret sig, bare fordi AI-arbejdere er involveret. Du skal stadig tage hensyn til andre faktorer såsom ydeevne, skalerbarhed og sikkerhed, men nu

er der også helt nye "bløde krav" at overveje. For eksempel er mange brugere imod at få deres private data brugt til at træne nye AI-modeller. Har du verificeret niveauet af privatlivsbeskyttelse, som modeludbyderen du bruger, tilbyder?

## AI-Arbejdere Som Mikroservices?

Når du læser om tilgangen med Mangfoldighed af Arbejdere, vil du måske bemærke nogle ligheder med Mikroservice-arkitektur. Begge lægger vægt på nedbrydningen af komplekse systemer i mindre, mere håndterbare og uafhængigt implementerbare enheder. Ligesom mikroservices er designet til at være løst koblede, fokuserede på specifikke forretningskapaciteter og kommunikerer gennem veldefinerede API'er, er AI-arbejdere designet til at være modulære, specialiserede i deres opgaver og interagere med hinanden gennem klare grænseflader og kommunikationsprotokoller.

Der er dog nogle vigtige forskelle at huske på. Mens mikroservices typisk implementeres som separate processer eller tjenester, der kører på forskellige maskiner eller containere, kan AI-arbejdere implementeres som selvstændige komponenter inden for en enkelt applikation eller som separate tjenester, afhængigt af dine specifikke krav og skaleringsbehov. Derudover involverer kommunikationen mellem AI-arbejdere ofte udveksling af rig, naturlig sprogbaseret information, såsom prompts, instruktioner og genereret indhold, snarere end de mere strukturerede dataformater, der almindeligvis bruges i mikroservices.

På trods af disse forskelle forbliver principperne om modularitet, løs kobling og klare kommunikationsgrænseflader centrale for begge mønstre. Ved at anvende disse principper på din AI-arbejder-arkitektur kan du skabe fleksible, skalerbare og vedligeholdelsesvenlige systemer, der udnytter AI's kraft til at løse komplekse problemer og levere værdi til dine brugere.

Tilgangen med Mangfoldighed af Arbejdere kan anvendes på tværs af forskellige domæner og applikationer, hvor man udnytter AI's kraft til at tackle komplekse opgaver og levere intelligente løsninger. Lad os udforske nogle konkrete eksempler på, hvordan AI-arbejdere kan anvendes i forskellige sammenhænge.

## Kontoadministration

Praktisk talt hver eneste selvstændig webapplikation har konceptet om en konto (eller bruger). I Olympia anvender vi en `AccountManager` AI-arbejder, der er programmeret til at kunne håndtere forskellige typer af ændringsanmodninger relateret til brugerkonti.

Dets direktiv lyder således:

```
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 indledende tilstand af kontoen produceret af `account.to_directive` er ganske enkelt en tekstbeskrivelse af kontoen, herunder relevante tilknyttede data såsom brugere, abonnementer osv.

Udvalget af funktioner tilgængelige for `AccountManager` giver den mulighed for at redigere brugerens abonnement, tilføje og fjerne AI-konsulenter og andre typer

betalte tilføjelser samt sende notifikations-e-mails til kontoens ejer. Ud over `finished`-funktionen kan den også `notify_human_administrator`, hvis den støder på en fejl under behandlingen eller har brug for anden form for assistance med en anmodning.

Bemærk, at i tilfælde af spørgsmål kan `AccountManager` vælge at søge i Olympias vidensbase, hvor den kan finde instruktioner om, hvordan man håndterer særtilfælde og enhver anden situation, hvor den er usikker på, hvordan den skal fortsætte.

# E-handelsapplikationer

Inden for e-handel kan AI-arbejdere spille en afgørende rolle i at forbedre brugeroplevelsen og optimere forretningsdriften. Her er nogle måder, hvorpå AI-arbejdere kan anvendes:

## Produktanbefalinger

En af de mest effektive anvendelser af AI-arbejdere inden for e-handel er generering af personlige produktanbefalinger. Ved at analysere brugeradfærd, købshistorik og præferencer kan disse arbejdere foreslå produkter, der er skræddersyet til hver enkelt brugers interesser og behov.

Nøglen til effektive produktanbefalinger er at udnytte en kombination af kollaborativ filtrering og indholdsbaseret filtrering. Kollaborativ filtrering ser på adfærden hos lignende brugere for at identificere mønstre og lave anbefalinger baseret på, hvad andre med lignende smag har købt eller syntes godt om. Indholdsbaseret filtrering fokuserer derimod på produkternes egenskaber og attributter og anbefaler varer, der deler lignende funktioner med dem, en bruger tidligere har vist interesse for.

Her er et forenklet eksempel på, hvordan du kan implementere en produktanbefalings-worker i Ruby, denne gang ved hjælp af en "Railway Oriented (ROP)" funktionel 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
```

Ruby-stilen med funktionel programmering, der bruges i eksemplet, er inspireret af F# og Rust. Du kan læse mere om det i min ven Chad Wooleys forklaring af teknikken hos GitLab

I dette eksempel tager `ProductRecommendationWorker` en bruger som input og genererer personlige produktanbefalinger ved at sende et værdiobjekt gennem en kæde af funktionelle trin. Lad os gennemgå hvert trin:

1. `ValidateUser.validate`: Dette trin sikrer, at brugeren er gyldig og berettiget til personlige anbefalinger. Det kontrollerer, om brugeren eksisterer, er aktiv og har de nødvendige data tilgængelige til at generere anbefalinger. Hvis valideringen fejler, returneres et fejlresultat, og kæden afbrydes tidligt.
2. `AnalyzeCurrentSession.analyze`: Hvis brugeren er gyldig, analyserer dette trin brugerens aktuelle browsing-session for at indsamle kontekstuel

information. Det ser på brugerens seneste interaktioner, såsom viste produkter, søgeforespørgsler og indhold i indkøbskurven, for at forstå deres aktuelle interesser og hensigt.

3. `CollaborativeFilter.filter`: Ved hjælp af *adfærden fra lignende brugere* anvender dette trin kollaborative filtreringsteknikker til at identificere produkter, som sandsynligvis vil interessere brugeren. Det tager højde for faktorer som købshistorik, bedømmelser og bruger-produkt-interaktioner for at generere et sæt af mulige anbefalinger.
4. `ContentBasedFilter.filter`: Dette trin forfiner yderligere kandidatanbefalingerne ved at anvende indholdsbaseret filtrering. Det sammenligner egenskaber og karakteristika for kandidatprodukterne med *brugerens præferencer og historiske data* for at vælge de mest relevante varer.
5. `ProductSelector.select`: Endelig vælger dette trin de bedste N produkter fra de filtrerede anbefalinger baseret på foruddefinerede kriterier, såsom relevansscore, popularitet eller andre forretningsregler. De udvalgte produkter returneres derefter som de endelige personlige anbefalinger.

Det smukke ved at bruge en funktionel Ruby-programmeringsstil her er, at det tillader os at kæde disse trin sammen på en klar og præcis måde. Hvert trin fokuserer på en specifik opgave og returnerer et `Result`-objekt, som enten kan være en succes (`ok`) eller en fejl (`err`). Hvis et trin støder på en fejl, afbrydes kæden tidligt, og fejlen videregives til det endelige resultat.

I `case`-sætningen til sidst laver vi mønstergenkendelse på det endelige resultat. Hvis resultatet er en fejl (`ProductRecommendationError`), logger vi fejlen ved hjælp af et værktøj som Honeybadger til overvågning og fejlfinding. Hvis resultatet er en succes (`ProductRecommendations`), udsender vi en `:new_recommendations`-begivenhed ved hjælp af Wisper pub/sub-biblioteket, hvor vi videresender brugeren og de genererede anbefalinger.

Ved at udnytte funktionelle programmeringsteknikker kan vi skabe en modulær og

vedligeholdelsesvenlig product recommendation worker. Hvert trin er selvstændigt og kan nemt testes, ændres eller udskiftes uden at påvirke det overordnede flow. Brugen af mønstergenkendelse og `Result`-klassen hjælper os med at håndtere fejl elegant og sikrer, at workeren fejler hurtigt, hvis et trin støder på et problem.

Dette er naturligvis et forenklet eksempel, og i en virkelig situation ville du skulle integrere med din e-handelsplatform, håndtere særtilfælde og endda dykke ned i implementeringen af anbefalingsalgoritmerne. Dog forbliver kerneprincipper om at opdele problemet i mindre trin og udnytte funktionelle programmeringsteknikker de samme.

## Svindelregistrering

Her er et forenklet eksempel på, hvordan du kan implementere en svindelregistrerings-worker ved hjælp af samme Railway Oriented Programming (ROP)-stil 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
```

Klassen `FraudDetection` er et *value object*, der indkapsler svigdetektionsstatus for en given transaktion. Den giver en struktureret måde at analysere og vurdere risikoen for svindel forbundet med en transaktion baseret på forskellige 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
```

Klassen `FraudDetection` har følgende attributter:

- `transaction`: En reference til den transaktion, der analyseres for svindel.
- `risk_factors`: Et array, der gemmer risikofaktorerne forbundet med transaktionen. Hver risikofaktor er repræsenteret som et hash, hvor nøglen er beskrivelsen af risikofaktoren, og værdien er sandsynligheden for svindel forbundet med den pågældende risikofaktor.

Metoden `add_risk_factor` gør det muligt at tilføje en risikofaktor til `risk_factors`-arrayet. Den tager to parametre: `description`, som er en streng, der beskriver risikofaktoren, og `probability`, som er et decimaltal, der repræsenterer sandsynligheden for svindel forbundet med den pågældende risikofaktor. Vi bruger en `case..in`-betingelse til at udføre simpel typevalidering.

Metoden `high_risk?`, som vil blive kontrolleret i slutningen af kæden, er en prædikatsmetode, der sammenligner `fraud_probability` (beregnet ved at summere sandsynlighederne for alle risikofaktorer) med `RISK_THRESHOLD`.

Klassen `FraudDetection` giver en ren og indkapslet måde at håndtere svigdetektion for en transaktion. Den tillader tilføjelse af flere risikofaktorer, hver med sin egen beskrivelse og sandsynlighed, og leverer en metode til at afgøre, om transaktionen anses for at være høj-risiko baseret på den beregnede svindelsandsynlighed. Klassen kan nemt integreres i et større svigdetektionssystem, hvor forskellige komponenter kan samarbejde om at vurdere og reducere risikoen for svigagtige transaktioner.

Endelig, eftersom dette trods alt er en bog om programmering ved hjælp af AI, er her et eksempel på implementering af klassen `CheckCustomerHistory`, der udnytter AI-behandling ved hjælp af mit Raix-biblioteks `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 eksempel definerer `CheckCustomerHistory` en `INSTRUCTION`-konstant, der giver specifikke instruktioner til AI-modellen om, hvordan kundens transaktionshistorik skal analyseres for potentielle svigindikationer via et systemdirektiv

`self.check`-metoden er en klassemetode, der initialiserer en ny instans af `CheckCustomerHistory` med `fraud_detection`-objektet og kalder `call`-metoden for at udføre analysen af kundehistorikken.

I `call`-metoden hentes kundens transaktionshistorik og formateres til et transskript, der sendes til AI-modellen. AI-modellen analyserer transaktionshistorikken baseret på de givne instruktioner og returnerer et sammendrag af sine fund.

Resultaterne tilføjes til `fraud_detection`-objektet, og det opdaterede `fraud_detection`-objekt returneres som et vellykket `Result`.

Ved at udnytte `ChatCompletion`-modulet kan `CheckCustomerHistory`-klassen anvende AI'ens kraft til at analysere kundens transaktionshistorik og identificere potentielle svigindikationer. Dette muliggør mere sofistikerede og tilpasningsdygtige svigdetektionsteknikker, da AI-modellen kan lære og tilpasse sig nye mønstre og anomalier over tid.

Den opdaterede `FraudDetectionWorker` og `CheckCustomerHistory`-klassen demonstrerer, hvordan AI-arbejdere kan integreres problemfrit og forbedre svigdetektionsprocessen med intelligent analyse og beslutningstagningsevner.

## Kundesentimentanalyse

Her er endnu et lignende eksempel på, hvordan du kan implementere en kundesentimentanalyse-worker. Meget mindre forklaring denne gang, da du burde være ved at forstå, hvordan denne programmeringsstil 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 eksempel omfatter trinnene i `CustomerSentimentAnalysisWorker` forbehandling af feedback (f.eks. fjernelse af støj, tokenisering), udførelse af sentimentanalyse for at bestemme den overordnede stemning (positiv, negativ eller neutral), udtrækning af nøglefraser og emner, identifikation af tendenser og mønstre samt generering af handlingsorienterede indsigter baseret på analysen.

# Sundhedsvæsenets anvendelser

Inden for sundhedssektoren kan AI-arbejdere assistere sundhedspersonale og forskere med forskellige opgaver, hvilket fører til forbedrede patientresultater og accelererede medicinske opdagelser. Nogle eksempler omfatter:

## Patientmodtagelse

AI-arbejdere kan effektivisere patientmodtagelsesprocessen ved at automatisere forskellige opgaver og yde intelligent assistance.

**Tidsbestilling**: AI-arbejdere kan håndtere tidsbestilling ved at forstå patienternes præferencer, tilgængelighed og deres medicinske behovs hastende karakter. De kan interagere med patienter gennem samtalebaserede grænseflader, guide dem gennem bookingprocessen og finde de mest passende tidspunkter baseret på patientens behov og sundhedspersonalets tilgængelighed.

**Indsamling af sygehistorie**: Under patientmodtagelsen kan AI-arbejdere hjælpe med at indsamle og dokumentere patientens sygehistorie. De kan føre interaktive dialoger med patienter og stille relevante spørgsmål om deres tidligere sygdomme, medicin, allergier og familiehistorie. AI-arbejderne kan bruge naturlig sprogbehandling til at fortolke og strukturere de indsamlede oplysninger og sikre, at de registreres nøjagtigt i patientens elektroniske patientjournal.

**Symptomvurdering og stratificering**: AI-arbejdere kan udføre indledende symptomvurderinger ved at spørge patienter om deres aktuelle symptomer, varighed, sværhedsgrad og eventuelle tilknyttede faktorer. Ved at udnytte medicinske videnbaser og maskinlæringsmodeller kan disse arbejdere analysere de givne oplysninger og generere foreløbige differentialdiagnoser eller anbefale passende næste trin, såsom at planlægge en konsultation hos en sundhedsudbyder eller foreslå selvhjælpsforanstaltninger.

**Forsikringsverifikation**: AI-arbejdere kan assistere med forsikringsverifikation under patientmodtagelsen. De kan indsamle patientens forsikringsoplysninger, kommunikere med forsikringsselskaber gennem API'er eller webtjenester og verificere dækningsberettigelse og ydelser. Denne automatisering hjælper med at strømline forsikringsverifikationsprocessen, reducere den administrative byrde og sikre nøjagtig informationsregistrering.

**Patientuddannelse og instruktioner**: AI-arbejdere kan forsyne patienter med relevant uddannelsesmateriale og instruktioner baseret på deres specifikke medicinske tilstande eller kommende procedurer. De kan levere personligt tilpasset indhold, besvare almindelige spørgsmål og give vejledning om forberedelser før konsultation, medicininstruktioner eller efterbehandlingspleje. Dette hjælper med at holde patienter informerede og engagerede gennem hele deres sundhedsrejse.

Ved at udnytte AI-arbejdere i patientmodtagelsen kan sundhedsorganisationer øge effektiviteten, reducere ventetider og forbedre den samlede patientoplevelse. Disse arbejdere kan håndtere rutineopgaver, indsamle nøjagtige oplysninger og yde personlig assistance, hvilket giver sundhedspersonalet mulighed for at fokusere på at levere pleje af høj kvalitet til patienterne.

## Patientrisiko-vurdering

AI-arbejdere kan spille en afgørende rolle i vurdering af patientrisiko ved at analysere forskellige datakilder og anvende avancerede analyseteknikker.

**Dataintegration**: AI-arbejdere kan indsamle og skabe mening i patientdata fra flere kilder, såsom elektroniske patientjournaler (EPJ), medicinske billeder, laboratorieresultater, wearables og sociale sundhedsdeterminanter. Ved at konsolidere disse oplysninger til en omfattende patientprofil kan AI-arbejdere give et holistisk billede af patientens helbredstilstand og risikofaktorer.

**Risikostratificering**: AI-arbejdere kan bruge prædiktive modeller til at stratificere patienter i forskellige risikokategorier baseret på deres individuelle karakteristika og

sundhedsdata. Denne risikostratificering gør det muligt for sundhedspersonalet at prioritere patienter, der kræver mere umiddelbar opmærksomhed eller intervention. For eksempel kan patienter, der identificeres som højrisiko for en bestemt tilstand, markeres til tættere overvågning, forebyggende foranstaltninger eller tidlig intervention.

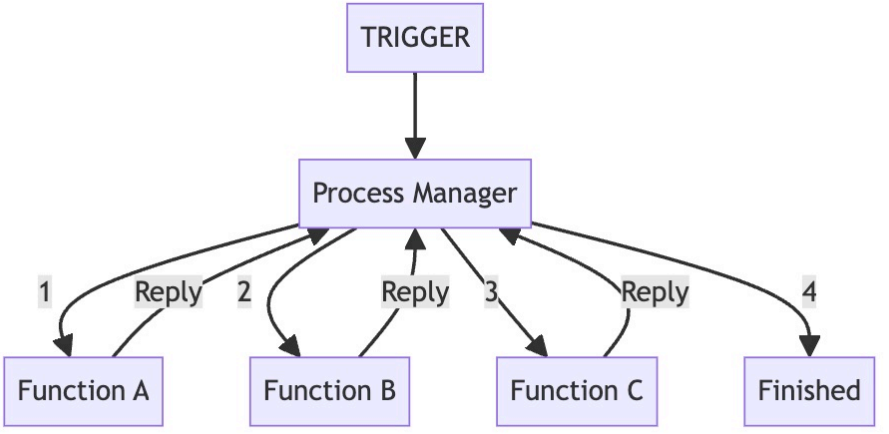
**Personlige risikoprofiler**: AI-arbejdere kan generere personlige risikoprofiler for hver patient, der fremhæver de specifikke faktorer, der bidrager til deres risikoscorer. Disse profiler kan omfatte indsigt i patientens livsstil, genetiske dispositioner, miljømæssige faktorer og sociale sundhedsdeterminanter. Ved at give en detaljeret nedbrydning af risikofaktorer kan AI-arbejdere hjælpe sundhedspersonalet med at skræddersy forebyggelsesstrategier og behandlingsplaner til individuelle patientbehov.

**Kontinuerlig risikoovervågning**: AI-arbejdere kan kontinuerligt overvåge patientdata og opdatere risikovurderinger i realtid. Efterhånden som nye oplysninger bliver tilgængelige, såsom ændringer i vitale tegn, laboratorieresultater eller medicinefterlevelse, kan AI-arbejdere genberegne risikoscorer og advare sundhedspersonalet om eventuelle væsentlige ændringer. Denne proaktive overvågning muliggør rettidige interventioner og justeringer af patientens behandlingsplaner.

**Klinisk beslutningsstøtte**: AI-arbejdere kan integrere resultater af risikovurderinger i kliniske beslutningsstøttesystemer og give sundhedspersonalet evidensbaserede anbefalinger og advarsler. For eksempel, hvis en patients risikoscore for en bestemt tilstand overstiger en vis tærskel, kan AI-arbejderen opfordre sundhedspersonalet til at overveje specifikke diagnostiske tests, forebyggende foranstaltninger eller behandlingsmuligheder baseret på kliniske retningslinjer og best practices.

Disse workers kan behandle store mængder patientdata, anvende avanceret analyse og generere handlingsorienterede indsigter til støtte for klinisk beslutningstagning. Dette fører i sidste ende til forbedrede patientresultater, reducerede sundhedsomkostninger og forbedret befolkningssundhedsstyring.

# AI Worker som Processtyring




I forbindelse med AI-drevne applikationer kan en worker designes til at fungere som en Processtyring, som beskrevet i bogen "Enterprise Integration Patterns" af Gregor Hohpe. En Processtyring er en central komponent, der opretholder processens tilstand og bestemmer de næste behandlingstrin baseret på mellemliggende resultater.

Når en AI-worker fungerer som Processtyring, modtager den en indgående besked, der initialiserer processen, kendt som *udløserbeskeden*. AI-workeren opretholder derefter processens udførelsestilstand (som en samtaleudskrift) og håndterer beskeden gennem en række behandlingstrin implementeret som værktøjsfunktioner, der kan være sekventielle eller parallelle, og kaldes efter dens skøn.

Hvis du bruger en klasse af AI-model som GPT-4, der ved, hvordan man udfører funktioner parallelt, kan din worker udføre flere trin samtidigt. Jeg må indrømme, at jeg ikke selv har prøvet det, og min mavefornemmelse siger, at resultaterne kan variere.

Efter hvert enkelt behandlingstrin returneres kontrollen tilbage til AI-workeren, hvilket giver den mulighed for at bestemme de(t) næste behandlingstrin baseret på den aktuelle tilstand og de opnåede resultater.

## Gem dine udløserbeskeder

Efter min erfaring er det klogt at implementere din udløserbesked som et databaseunderstøttet objekt. På den måde identificeres hver procesinstans af en unik primærnøgle og giver dig et sted at gemme den tilstand, der er forbundet med udførelsen, herunder AI'ens samtaleudskrift.

Her er for eksempel en forenklet version af Olympias `AccountChange`-modelklasse, som repræsenterer en anmodning om at foretage en ændring i en brugers 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 udløserbesked, der igangsætter en proces til håndtering af kontoændringsanmodningen. Bemærk, hvordan den broadcastes til Olympias Wisper-baserede pub/sub-undersystem efter oprettelsestransaktionen er færdig med at blive gennemført.

At gemme udløserbeskeden i databasen på denne måde giver en vedvarende registrering af hver kontoændringsanmodning. Hver instans af klassen `AccountChange` tildeles en unik primærnøgle, hvilket muliggør nem identifikation og sporing af individuelle anmodninger. Dette er særligt nyttigt i forbindelse med revisionslogning, da det gør det muligt for systemet at opretholde en historisk oversigt over alle kontoændringer, herunder hvornår de blev anmodet, hvilke ændringer der blev anmodet om, og den aktuelle status for hver anmodning.

I det givne eksempel indeholder klassen `AccountChange` felter som `description` til at registrere detaljerne for den ønskede ændring, `state` til at repræsentere anmodningens aktuelle tilstand (f.eks. anmodet, fuldført, kræver_manuel_gennemgang), og `transcript` til at gemme AI'ens samtaleudskrift relateret til anmodningen. Feltet `description` er den faktiske prompt, der bruges til at igangsætte den første chat-færdiggørelse med AI'en. At gemme disse data giver værdifuld kontekst og muliggør bedre sporing og analyse af kontoændringsprocessen.

At gemme udløserbeskeder i databasen muliggør robust fejlhåndtering og genopretning. Hvis der opstår en fejl under behandlingen af en kontoændringsanmodning, markerer systemet anmodningen som fejlet og overfører den til en tilstand, der kræver menneskelig indgriben. Dette sikrer, at ingen anmodninger går tabt eller bliver glemt, og at eventuelle problemer kan håndteres og løses korrekt.

AI-workeren fungerer som en Process Manager og giver et centralt kontrolpunkt samt kraftfulde muligheder for procesrapportering og fejlfinding. Det er dog vigtigt at bemærke, at brugen af en AI-worker som Process Manager for hvert workflow-scenarie i din applikation kan være overdrevet.

# Integration af AI-Workers I Din Applikationsarkitektur

Når man integrerer AI-workers i sin applikationsarkitektur, er der flere tekniske overvejelser, der skal adresseres for at sikre en gnidningsfri integration og effektiv kommunikation mellem AI-workers og andre applikationskomponenter. Dette afsnit behandler vigtige aspekter af design af disse grænseflader, håndtering af dataflow og styring af AI-workers' livscyklus.

## Design af Klare Grænseflader og Kommunikationsprotokoller

For at facilitere en problemfri integration mellem AI-workers og andre applikationskomponenter er det afgørende at definere klare grænseflader og kommunikationsprotokoller. Overvej følgende tilgange:

**API-baseret Integration**: Eksponér AI-workers' funktionalitet gennem veldefinerede API'er, såsom RESTful endpoints eller GraphQL-skemaer. Dette gør det muligt for andre

komponenter at interagere med AI-workers ved hjælp af standard HTTP-anmodninger og -svar. API-baseret integration giver en klar kontrakt mellem AI-workers og de forbrugende komponenter, hvilket gør det lettere at udvikle, teste og vedligeholde integrationspunkterne.

**Beskedbaseret Kommunikation**: Implementér beskedbaserede kommunikationsmønstre, såsom beskedkøer eller publish-subscribe-systemer, for at muliggøre asynkron interaktion mellem AI-workers og andre komponenter. Denne tilgang afkobler AI-workers fra resten af applikationen, hvilket muliggør bedre skalerbarhed, fejltolerance og løs kobling. Beskedbaseret kommunikation er særligt nyttig, når behandlingen udført af AI-workers er tidskrævende eller ressourceintensiv, da det tillader andre dele af applikationen at fortsætte uden at vente på, at AI-workers færdiggør deres opgaver.

**Hændelsesdrevet Arkitektur**: Design dit system omkring hændelser og udløsere, der aktiverer AI-workers, når specifikke betingelser er opfyldt. AI-workers kan abonnere på relevante hændelser og reagere i overensstemmelse hermed, udføre deres designerede opgaver når hændelserne opstår. Hændelsesdrevet arkitektur muliggør realtidsbehandling og tillader AI-workers at blive aktiveret efter behov, hvilket reducerer unødvendigt ressourceforbrug. Denne tilgang er velegnet til scenarier, hvor AI-workers skal reagere på specifikke handlinger eller ændringer i applikationens tilstand.

## Håndtering af Dataflow og Synkronisering

Når du integrerer AI-workers i din applikation, er det afgørende at sikre et gnidningsfrit dataflow og opretholde datakonsistens mellem AI-workers og andre komponenter. Overvej følgende aspekter:

**Dataforberedelse**: Før data fødes ind i AI-workers, kan det være nødvendigt at udføre forskellige dataforberedelsesopgaver, såsom rensning, formatering og/eller transformation af inputdata. Du vil ikke kun sikre, at AI-workers kan behandle effektivt, men også sikre, at du ikke spilder tokens ved at give opmærksomhed til

information, som workeren måske anser for ubrugelig i bedste fald og distraherende i værste fald. Dataforberedelse kan omfatte opgaver som fjernelse af støj, håndtering af manglende værdier eller konvertering af datatyper.

**Datapersistens**: Hvordan vil du gemme og bevare de data, der flyder ind og ud af AI-workers? Overvej faktorer som datavolumen, forespørgselsmønstre og skalerbarhed. Har du behov for at gemme AI'ens udskrift som en afspejling af dens "tankeproces" til revisions- eller fejlfindingsformål, eller er det tilstrækkeligt at have en registrering af resultaterne alene?

**Datahentning**: At hente de data, som workers har brug for, kan involvere databaseforespørgsler, læsning fra filer eller adgang til eksterne API'er. Sørg for at overveje latenstid og hvordan AI-workers vil have adgang til de mest opdaterede data. Har de brug for fuld adgang til din database, eller bør du definere omfanget af deres adgang snævert i forhold til deres opgaver? Hvad med skalering? Overvej caching-mekanismer for at forbedre ydeevnen og reducere belastningen på de underliggende datakilder.

**Datasynkronisering**: Når flere komponenter, herunder AI-workers, tilgår og ændrer delte data, er det vigtigt at implementere passende synkroniseringsmekanismer for at opretholde datakonsistens. Databaselåsningsstrategier, såsom optimistisk eller pessimistisk låsning, kan hjælpe dig med at forhindre konflikter og sikre dataintegritet. Implementer transaktionsstyringsteknikker for at gruppere relaterede dataoperationer og opretholde ACID-egenskaberne (atomaritet, konsistens, isolation og holdbarhed)

**Fejlhåndtering og Genopretning**: Implementer robuste fejlhåndterings- og genopretningsmekanismer til at håndtere datarelaterede problemer, der kan opstå under dataflowprocessen. Håndter undtagelser elegant og giv meningsfulde fejlmeddelelser til hjælp ved fejlfinding. Implementer gentagelsesmekanismer og fallback-strategier til at håndtere midlertidige fejl eller netværksforstyrrelser. Definer klare procedurer for datagenopretning og -gendannelse i tilfælde af datakorruption eller -tab.

Ved omhyggelig design og implementering af dataflow- og synkroniseringsmekanismer

kan du sikre, at dine AI-workers har adgang til præcise, konsistente og opdaterede data. Dette gør dem i stand til at udføre deres opgaver effektivt og producere pålidelige resultater.

## Håndtering af AI-Workers' Livscyklus

Udvikl en standardiseret proces til initialisering og konfiguration af AI-workers. Jeg foretrækker frameworks, der standardiserer, hvordan du definerer indstillinger såsom modelnavne, systemdirektiver og funktionsdefinitioner. Sørg for, at initialiseringsprocessen er automatiseret og reproducerbar for at lette implementering og skalering.

Implementer omfattende overvågnings- og logningsmekanismer til at spore AI-workers' sundhed og ydeevne. Indsaml målinger såsom ressourceforbrug, behandlingstid, fejlrater og gennemløb. Brug centraliserede logningssystemer som ELK-stack (Elasticsearch, Logstash, Kibana) til at aggregere og analysere logs fra flere AI-workers.

Byg fejltolerance og robusthed ind i AI-worker-arkitekturen. Implementer fejlhåndterings- og genopretningsmekanismer til elegant at håndtere fejl eller undtagelser. Store Sprogmodeller er stadig cutting-edge teknologi; udbydere har tendens til ofte at gå ned på uventede tidspunkter. Brug gentagelsesmekanismer og kredsløbsafbrydere for at forhindre kaskaderende fejl.

# Sammensættelighed og Orkestrering af AI-Workers

En af de vigtigste fordele ved AI-worker-arkitekturen er dens sammensættelighed, som gør det muligt at kombinere og orkestrere flere AI-workers til at løse komplekse problemer. Ved at nedbryde en større opgave i mindre, mere håndterbare delopgaver,

der hver håndteres af en specialiseret AI-worker, kan du skabe kraftfulde og fleksible systemer. I dette afsnit vil vi udforske forskellige tilgange til at sammensætte og orkestrere "en mangfoldighed" af AI-workers.

## Sammenkædning af AI-Workers til Flertrinsprocedurer

I mange scenarier kan en kompleks opgave nedbrydes i en serie af sekventielle trin, hvor outputtet fra én AI-worker bliver inputtet for den næste. Denne sammenkædning af AI-workers skaber en flertrinsprocedure eller pipeline. Hver AI-worker i kæden fokuserer på en specifik delopgave, og det endelige output er resultatet af den kombinerede indsats fra alle workers.

Lad os betragte et eksempel i konteksten af en Ruby on Rails-applikation til behandling af brugergenereret indhold. Arbejdsgangen involverer følgende trin, som indrømmet sandsynligvis hver især er for simple til at være værd at nedbryde på denne måde i virkelige anvendelser, men de gør eksemplet lettere at forstå:

1. **Tekstrensning**: En AI-worker ansvarlig for at fjerne HTML-tags, konvertere tekst til små bogstaver og håndtere Unicode-normalisering.

2. **Sprogdetektering**: En AI-worker der identificerer sproget i den rensede tekst.

3. **Sentimentanalyse**: En AI-worker der bestemmer sentimentet (positiv, negativ eller neutral) i teksten baseret på det detekterede sprog.

4. **Indholdskateogorisering**: En AI-worker der klassificerer teksten i foruddefinerede kategorier ved hjælp af naturlig sprogbehandlingsteknikker.

Her er et meget forenklet eksempel på, hvordan du kan sammenkæde disse AI-workers ved hjælp af 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 eksempel initialiserer `ContentProcessor`-klassen med råteksten og kæder AI-arbejderne sammen i `process`-metoden. Hver AI-arbejder udfører sin specifikke opgave og sender resultatet videre til den næste arbejder i kæden. Det endelige output er et hash, der indeholder den rensede tekst, det detekterede sprog, sentiment og indholdskategori.

## Parallel behandling for uafhængige AI-arbejdere

I det foregående eksempel er AI-arbejderne kædet sekventielt sammen, hvor hver arbejder behandler teksten og sender resultatet videre til den næste arbejder. Men hvis du har flere AI-arbejdere, der kan operere uafhængigt på samme input, kan du optimere arbejdsgangen ved at behandle dem parallelt.

I det givne scenarie kan `LanguageDetectionWorker`, `SentimentAnalysisWorker` og `CategorizationWorker` alle behandle den rensede tekst uafhængigt, når tekstrensningen er udført af `TextCleanupWorker`. Ved at køre disse arbejdere parallelt kan du potentielt reducere den samlede behandlingstid og forbedre effektiviteten af din arbejdsgang.

For at opnå parallel behandling i Ruby kan du udnytte samtidighedsteknikker såsom tråde eller asynkron programmering. Her er et eksempel på, hvordan du kan modificere

`ContentProcessor`-klassen til at behandle de sidste tre arbejdere parallelt ved hjælp af tråde:

```
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 optimerede version bruger vi `concurrent-ruby`-biblioteket til at oprette `Concurrent::Future`-objekter for hver af de uafhængige AI-arbejdere. En `Future` repræsenterer en beregning, der vil blive udført asynkront i en separat tråd.

Efter tekstrensnings-trinnet opretter vi tre `Future`-objekter: `language_future`, `sentiment_future` og `category_future`. Hver `Future` udfører sin tilsvarende

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

Ved at kalde `value`-metoden på hver `Future`, venter vi på at beregningen færdiggøres og henter resultatet. `value`-metoden blokerer indtil resultatet er tilgængeligt, hvilket sikrer at alle parallelle arbejdere har afsluttet deres behandling før vi fortsætter.

Til sidst konstruerer vi output-hashen med den rensede tekst og resultaterne fra de parallelle arbejdere, præcis som i det oprindelige eksempel.

Ved at behandle de uafhængige AI-arbejdere parallelt kan du potentielt reducere den samlede behandlingstid sammenlignet med at køre dem sekventielt. Denne optimering er særligt fordelagtig når der arbejdes med tidskrævende opgaver eller ved behandling af store datamængder.

Det er dog vigtigt at bemærke, at de faktiske ydelsesforbedringer afhænger af forskellige faktorer, såsom kompleksiteten af hver arbejder, de tilgængelige systemressourcer og overhead fra trådhåndtering. Det er altid god praksis at lave benchmark og profilering af din kode for at bestemme det optimale niveau af parallelisme for dit specifikke anvendelsestilfælde.

Derudover skal du, når du implementerer parallel behandling, være opmærksom på eventuelle delte ressourcer eller afhængigheder mellem arbejderne. Sørg for at arbejderne kan operere uafhængigt uden konflikter eller kapløbstilstande. Hvis der er afhængigheder eller delte ressourcer, kan du blive nødt til at implementere passende synkroniseringsmekanismer for at opretholde dataintegritet og undgå problemer som deadlocks eller inkonsistente resultater.

### Rubys Global Interpreter Lock og Asynkron Behandling

Det er vigtigt at forstå implikationerne af Rubys Global Interpreter Lock (GIL) når man overvejer asynkron trådbaseret behandling i Ruby.

GIL'en er en mekanisme i Rubys fortolker, der sikrer at kun én tråd kan udføre Ruby-kode ad gangen, selv på multicore-processorer. Dette betyder, at selvom flere tråde kan oprettes og håndteres inden for en Ruby-proces, kan kun én tråd aktivt udføre Ruby-kode på et givet tidspunkt.

GIL'en er designet til at forenkle implementeringen af Ruby-fortolkeren og give trådsikkerhed for Rubys interne datastrukturer. Den begrænser dog også muligheden for ægte parallel udførelse af Ruby-kode.

Når du bruger tråde i Ruby, såsom med `concurrent-ruby`-biblioteket eller den indbyggede `Thread`-klasse, er trådene underlagt GIL'ens begrænsninger. GIL'en tillader hver tråd at udføre Ruby-kode i en kort tidsperiode, før den skifter til en anden tråd, hvilket skaber illusionen af samtidig udførelse.

På grund af GIL'en forbliver den faktiske udførelse af Ruby-kode dog sekventiel. Mens én tråd udfører Ruby-kode, er andre tråde i praksis sat på pause, ventende på deres tur til at erhverve GIL'en og udføre kode.

Dette betyder, at trådbaseret asynkron behandling i Ruby er mest effektiv for I/O-bundne opgaver, såsom at vente på svar fra eksterne API'er (som f.eks. eksternt hostede store sprogmodeller) eller udføre fil-I/O-operationer. Når en tråd møder en I/O-operation, kan den frigive GIL'en, hvilket tillader andre tråde at udføre kode mens der ventes på at I/O'en færdiggøres.

På den anden side kan GIL'en for CPU-bundne opgaver, såsom intensive beregninger eller langvarig AI-arbejder-behandling, begrænse de potentielle ydelsesgevinster ved trådbaseret parallelisme. Siden kun én tråd kan udføre Ruby-kode ad gangen, vil den samlede udførelsestid måske ikke blive væsentligt reduceret sammenlignet med sekventiel behandling.

For at opnå ægte parallel udførelse af CPU-bundne opgaver i Ruby kan du blive nødt til at udforske alternative tilgange, såsom:

- Brug af procesbaseret parallelisme med flere Ruby-processer, der hver kører på en separat CPU-kerne.
- Udnyttelse af eksterne biblioteker eller frameworks, der tilbyder native udvidelser eller grænseflader til sprog uden en GIL, såsom C eller Rust.,
- Anvendelse af distribuerede beregningsframeworks eller meddelelseskøer til at fordele opgaver på tværs af flere maskiner eller processer.

Det er afgørende at overveje karakteren af dine opgaver og de begrænsninger, som GIL'en pålægger, når du designer og implementerer asynkron behandling i Ruby. Mens trådbaseret asynkron behandling kan give fordele for I/O-bundne opgaver, tilbyder den måske ikke væsentlige ydelsesforbedringer for CPU-bundne opgaver på grund af GIL'ens begrænsninger.

## Ensemble-teknikker for Forbedret Nøjagtighed

Ensemble-teknikker involverer kombinationen af output fra flere AI-arbejdere for at forbedre systemets overordnede nøjagtighed eller robusthed. I stedet for at stole på en enkelt AI-arbejder, udnytter ensemble-teknikker den kollektive intelligens fra flere arbejdere til at træffe mere informerede beslutninger.

Ensembler er særligt vigtige, hvis forskellige dele af din arbejdsgang fungerer bedst med forskellige AI-modeller, hvilket er mere almindeligt, end du måske tror. Kraftfulde modeller som GPT-4 er ekstremt dyre sammenlignet med mindre avancerede open source-alternativer og er sandsynligvis ikke nødvendige for hvert eneste arbejdstrin i din applikation.

En almindelig ensemble-teknik er flertalsafstemning, hvor flere AI-arbejdere uafhængigt behandler det samme input, og det endelige output bestemmes af

flertallets konsensus. Denne tilgang kan hjælpe med at reducere påvirkningen af individuelle arbejderfejl og forbedre systemets generelle pålidelighed.

Lad os se på et eksempel, hvor vi har tre AI-arbejdere til sentimentanalyse, der hver især bruger en forskellig model eller er udstyret med forskellige kontekster. Vi kan kombinere deres output ved hjælp af flertalsafstemning for at bestemme den endelige sentimentforudsigelse.

```
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 eksempel initialiserer klassen `SentimentAnalysisEnsemble`. med teksten og aktiverer tre forskellige AI-arbejdere til sentimentanalyse. Metoden `analyze` indsamler forudsigelserne fra hver arbejder og bestemmer det dominerende sentiment ved hjælp af metoderne `group_by` og `max_by`. Det endelige output er det sentiment, der modtager flest stemmer fra ensemblet af arbejdere

Ensembler er helt klart et tilfælde, hvor det kan være værd at eksperimentere med parallelisme.

## Dynamisk udvælgelse og aktivering af AI-arbejdere

I nogle, hvis ikke de fleste tilfælde, kan den specifikke AI-arbejder, der skal aktiveres, afhænge af kørselstidsbetingelser eller brugerinput. Dynamisk udvælgelse og aktivering af AI-arbejdere giver fleksibilitet og tilpasningsevne i systemet.

Du kan måske blive fristet til at forsøge at passe meget funktionalitet ind i en enkelt AI-arbejder ved at give den mange funktioner og en stor kompliceret prompt, der forklarer, hvordan man bruger dem. Modstå fristelsen, stol på mig. En af grundene til, at den tilgang, vi diskuterer i dette kapitel, kaldes "Mangfoldighed af Arbejdere", er for at minde os om, at det er ønskværdigt at have mange specialiserede arbejdere, der hver især udfører deres egen lille opgave i den større sags tjeneste.

For eksempel kan man overveje en chatbot-applikation, hvor forskellige AI-arbejdere er ansvarlige for at håndtere forskellige typer af brugerforespørgsler. Baseret på brugerens input vælger applikationen dynamisk den passende AI-arbejder til at behandle forespørgslen.

```
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 eksempel modtager `ChatbotController` en brugerforespørgsel gennem `process_query`-handlingen. Den bruger først en `QueryClassifierWorker` til at bestemme forespørgslens type. Baseret på den klassificerede forespørgselstype vælger controlleren dynamisk den passende AI-worker til at generere svaret. Denne dynamiske udvælgelse gør det muligt for chatbotten at håndtere forskellige typer forespørgsler og dirigere dem til de relevante AI-workers.

Da arbejdet med `QueryClassifierWorker` er relativt enkelt og ikke kræver meget kontekst eller funktionsdefinitioner, kan du sandsynligvis implementere det ved hjælp af en ultrahurtig lille LLM som `mistralai/mixtral-8x7b-instruct:nitro`. Den har kapaciteter, der kommer tæt på GPT-4-niveau på mange opgaver, og på tidspunktet hvor jeg skriver dette, kan Groq levere den med en imponerende hastighed på 444 tokens i sekundet.

## Kombination af Traditionel NLP med LLM'er

Mens Store Sprogmodeller (LLM'er) har revolutioneret området inden for naturlig sprogbehandling (NLP), og tilbyder uovertruffen alsidighed og ydeevne på tværs af en bred vifte af opgaver, er de ikke altid den mest effektive eller omkostningseffektive løsning på ethvert problem. I mange tilfælde kan kombinationen af traditionelle NLP-teknikker med LLM'er føre til mere optimerede, målrettede og økonomiske tilgange til at løse specifikke NLP-udfordringer.

Tænk på LLM'er som schweizerknive inden for NLP - utroligt alsidige og kraftfulde, men ikke nødvendigvis det bedste værktøj til enhver opgave. Nogle gange kan

et dedikeret værktøj som en proptrækker eller en dåseåbner være mere effektivt til en specifik opgave. På samme måde kan traditionelle NLP-teknikker, såsom dokumentklyngedannelse, emneidentifikation og klassificering, ofte give mere målrettede og omkostningseffektive løsninger til visse aspekter af din NLP-pipeline.

En af de vigtigste fordele ved traditionelle NLP-teknikker er deres beregningsmæssige effektivitet. Disse metoder, som ofte er baseret på enklere statistiske modeller eller regelbaserede tilgange, kan behandle store mængder tekstdata meget hurtigere og med mindre beregningsoverhead sammenlignet med LLM'er. Dette gør dem særligt velegnede til opgaver, der involverer analyse og organisering af store dokumentsamlinger, såsom klyngedannelse af lignende artikler eller identifikation af nøgleemner inden for en samling af tekster.

Desuden kan traditionelle NLP-teknikker ofte opnå høj nøjagtighed og præcision for specifikke opgaver, især når de trænes på domænespecifikke datasæt. For eksempel kan en velindstillet dokumentklassifikator, der bruger traditionelle maskinlæringsalgoritmer som Support Vector Machines (SVM) eller Naive Bayes, præcist kategorisere dokumenter i foruddefinerede kategorier med minimal beregningsomkostning.

LLM'er skinner dog virkelig igennem, når det kommer til opgaver, der kræver en dybere forståelse af sprog, kontekst og ræsonnement. Deres evne til at generere sammenhængende og kontekstuelt relevant tekst, besvare spørgsmål og opsummere lange passager er uovertruffen af traditionelle NLP-metoder. LLM'er kan effektivt håndtere komplekse sproglige fænomener, såsom tvetydighed, koreference og idiomatiske udtryk, hvilket gør dem uvurderlige til opgaver, der kræver naturlig sproggenerering eller forståelse.

Den virkelige styrke ligger i at kombinere traditionelle NLP-teknikker med LLM'er for at skabe hybride tilgange, der udnytter styrkerne ved begge. Ved at bruge traditionelle NLP-metoder til opgaver som dokumentforbehandling, klyngedannelse og emneekstraktion kan du effektivt organisere og strukturere dine tekstdata. Denne strukturerede information kan derefter fødes ind i LLM'er til mere avancerede opgaver,

såsom generering af sammendrag, besvarelse af spørgsmål eller oprettelse af omfattende rapporter.

Lad os for eksempel overveje et anvendelsestilfælde, hvor du ønsker at generere en tendensrapport for et specifikt domæne baseret på et stort korpus af individuelle tendensdokumenter. I stedet for udelukkende at stole på LLM'er, som kan være beregningsmæssigt dyre og tidskrævende til behandling af store mængder tekst, kan du anvende en hybrid tilgang:

1. Brug traditionelle NLP-teknikker, såsom emnemodellering (f.eks. Latent Dirichlet Allocation) eller klyngedannelsesalgoritmer (f.eks. K-means), til at gruppere lignende tendensdokumenter sammen og identificere nøgletemaer og emner inden for korpusset.
2. Før de grupperede dokumenter og identificerede emner ind i en LLM, og udnyt dens overlegne sprogforståelse og genererende egenskaber til at skabe sammenhængende og informative sammendrag for hver klynge eller emne.
3. Brug endelig LLM'en til at generere en omfattende tendensrapport ved at kombinere de individuelle sammendrag, fremhæve de mest betydningsfulde tendenser og give indsigt og anbefalinger baseret på den samlede information.

Ved at kombinere traditionelle NLP-teknikker med LLM'er på denne måde kan du effektivt behandle store mængder tekstdata, udtrække meningsfuld indsigt og generere rapporter af høj kvalitet, samtidig med at du optimerer beregningsressourcer og omkostninger.

Når du går i gang med dine NLP-projekter, er det afgørende at evaluere de specifikke krav og begrænsninger for hver opgave grundigt og overveje, hvordan traditionelle NLP-metoder og LLM'er kan udnyttes sammen for at opnå de bedste resultater. Ved at kombinere effektiviteten og præcisionen fra traditionelle teknikker med alsidigheden og styrken fra LLM'er kan du skabe yderst effektive og økonomiske NLP-løsninger, der skaber værdi for dine brugere og interessenter.

# Brug af værktøjer

Inden for AI-drevet applikationsudvikling er konceptet "værktøjsbrug" eller "funktionskald" blevet en kraftfuld teknik, der gør det muligt for din LLM at forbinde sig til eksterne værktøjer, API'er, funktioner, databaser og andre ressourcer. Denne tilgang muliggør et rigere sæt af adfærdsmønstre end blot at outputte tekst og mere dynamiske interaktioner mellem dine AI-komponenter og resten af din applikations økosystem. Som vi vil undersøge i dette kapitel, giver værktøjsbrug dig også muligheden for at få din AI-model til at generere data på strukturerede måder.

## Hvad er værktøjsbrug?

Værktøjsbrug, også kendt som funktionskald, er en teknik, der gør det muligt for udviklere at specificere en liste af funktioner, som en LLM kan interagere med under genereringsprocessen. Disse værktøjer kan variere fra simple hjælpefunktioner til

komplekse API'er eller databaseforespørgsler. Ved at give LLM'en adgang til disse værktøjer kan udviklere udvide modellens muligheder og gøre den i stand til at udføre opgaver, der kræver ekstern viden eller handlinger.

**Figur 8. Eksempel på en funktionsdefinition for en AI-medarbejder, der 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
```

Hovedidéen bag værktøjsanvendelse er at give LLM'en mulighed for dynamisk at vælge og udføre de relevante værktøjer baseret på brugerens input eller den aktuelle opgave. I stedet for udelukkende at være afhængig af modellens forudtrænede viden, giver værktøjsanvendelse LLM'en mulighed for at udnytte eksterne ressourcer til at generere mere præcise, relevante og handlingsorienterede svar. Værktøjsanvendelse gør teknikker

som RAG (Retrieval Augmented Generation) meget lettere at implementere, end de ellers ville være.

> Bemærk, at medmindre andet er angivet, antager denne bog, at din AI-model ikke har adgang til indbyggede serverside-værktøjer. Alle værktøjer, du ønsker at stille til rådighed for din AI, skal eksplicit erklæres af dig i hver API-anmodning, med bestemmelser for deres udførelse, hvis og når din AI fortæller dig, at den ønsker at bruge det pågældende værktøj i sit svar.

## Potentialet i Værktøjsanvendelse

Værktøjsanvendelse åbner op for en bred vifte af muligheder for AI-drevne applikationer. Her er nogle eksempler på, hvad der kan opnås med værktøjsanvendelse:

1. **Chatbots og Virtuelle Assistenter**: Ved at forbinde en LLM til eksterne værktøjer kan chatbots og virtuelle assistenter udføre mere komplekse opgaver, såsom at hente information fra databaser, udføre API-kald eller interagere med andre systemer. For eksempel kunne en chatbot bruge et CRM-værktøj til at ændre status på en handel baseret på brugerens anmodning.
2. **Dataanalyse og Indsigter**: LLM'er kan forbindes til dataanalyseværktøjer eller biblioteker for at udføre avancerede databehandlingsopgaver. Dette gør det muligt for applikationer at generere indsigter, udføre komparative analyser eller give datadrevne anbefalinger baseret på brugerforespørgsler.
3. **Søgning og Informationshentning**: Værktøjsanvendelse giver LLM'er mulighed for at interagere med søgemaskiner, vektordatabaser eller andre informationshentningssystemer. Ved at omdanne brugerforespørgsler til søgeforespørgsler kan LLM'en hente relevant information fra flere kilder og give omfattende svar på brugerspørgsmål.

4. **Integration med Eksterne Tjenester**: Værktøjsanvendelse muliggør problemfri integration mellem AI-drevne applikationer og eksterne tjenester eller API'er. For eksempel kunne en LLM interagere med et vejr-API for at give realtids vejropdateringer eller et oversættelses-API for at generere flersprogede svar.

# Arbejdsgangen for Værktøjsanvendelse

Arbejdsgangen for værktøjsanvendelse involverer typisk fire hovedtrin:

1. Inkluder funktionsdefinitioner i din anmodningskontekst
2. Dynamisk (eller eksplicit) værktøjsvalg
3. Udførelse af funktion(er)
4. Valgfri fortsættelse af den oprindelige prompt

Lad os gennemgå hvert af disse trin i detaljer.

## Inkluder funktionsdefinitioner i din anmodningskontekst

AI'en ved, hvilke værktøjer den har til rådighed, fordi du giver den en liste som en del af din completion-anmodning (typisk defineret som funktioner ved hjælp af en variant af JSON-skema).

Den præcise syntaks for værktøjsdefinition er modelspecifik.

Sådan definerer du en `get_weather`-funktion 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 sådan definerer du den samme funktion for GPT-4, hvor du sender den som værdi til `tools`-parameteren:

```
{
    "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"],
```

```
    },
}
```

> Næsten det samme, bortset fra at det er anderledes uden nogen åbenlys grund! Hvor irriterende.

Funktionsdefinitioner angiver navn, beskrivelse og inputparametre. Inputparametre kan defineres yderligere ved hjælp af attributter såsom enums til at begrænse de acceptable værdier og ved at specificere, om en parameter er påkrævet eller ej.

Ud over de egentlige funktionsdefinitioner kan du også inkludere instruktioner eller kontekst for, hvorfor og hvordan funktionen skal bruges i systemdirektivet.

For eksempel indeholder mit Websøgningsværktøj i Olympia dette systemdirektiv, som minder AI'en om, at den har de nævnte værktøjer til rådighed:

```
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.
```

At give detaljerede beskrivelser anses for at være den vigtigste faktor for værktøjets ydeevne. Dine beskrivelser bør forklare alle detaljer om værktøjet, herunder:

- Hvad værktøjet gør
- Hvornår det bør bruges (og hvornår det ikke bør)
- Hvad hver parameter betyder, og hvordan den påvirker værktøjets adfærd
- Alle vigtige forbehold eller begrænsninger, der gælder for værktøjets implementering

Jo mere kontekst du kan give AI'en om dine værktøjer, jo bedre vil den være til at beslutte hvornår og hvordan de skal bruges. For eksempel anbefaler Anthropic mindst 3-4 sætninger per værktøjsbeskrivelse for deres Claude 3-serie, og flere hvis værktøjet er komplekst.

Det er ikke nødvendigvis intuitivt, men beskrivelser anses også for at være vigtigere end eksempler. Selvom du kan inkludere eksempler på, hvordan et værktøj bruges i dets beskrivelse eller i den medfølgende prompt, er dette mindre vigtigt end at have en klar og omfattende forklaring af værktøjets formål og parametre. Tilføj kun eksempler, efter du har udarbejdet beskrivelsen fuldt ud.

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

```
{
  "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 nogle modeller problemer med at håndtere indlejrede funktionsspecifikationer og komplekse output-datatyper som arrays, dictionaries osv. Men i teorien burde du kunne levere JSON Schema-specifikationer i vilkårlig dybde!

## Dynamisk Værktøjsvalg

Når du udfører en chat-færdiggørelse, der inkluderer værktøjsdefinitioner, vælger LLM'en dynamisk det mest passende værktøj(er) at bruge og genererer de nødvendige inputparametre for hvert værktøj.

I praksis er AI'ens evne til at kalde *præcis* den rigtige funktion og *præcis* følge din specifikation for inputs ikke altid pålidelig. At sætte temperature-hyperparameteren helt ned til 0.0 hjælper meget, men efter min erfaring vil du stadig opleve lejlighedsvise fejl. Disse fejl omfatter hallucinerede funktionsnavne, fejlnavngivne eller helt manglende inputparametre. Parametre overføres som JSON, hvilket betyder, at du nogle gange vil se fejl forårsaget af afkortet, fejlciteret eller på anden måde ødelagt JSON.

Selvhelende Data-mønstre kan hjælpe med at automatisk rette funktionskald, der går i stykker på grund af syntaksfejl.

## Tvunget (også kendt som Eksplicit) Værktøjsvalg

Nogle modeller giver dig mulighed for at tvinge kald af en bestemt funktion som en parameter i forespørgslen. Ellers er det helt op til AI'ens skøn, om funktionen skal kaldes eller ej.

Evnen til at tvinge et funktionskald er afgørende i visse scenarier, hvor du ønsker at sikre, at et specifikt værktøj eller funktion udføres, uanset AI'ens dynamiske udvælgelsesproces. Der er flere grunde til, at denne funktion er vigtig:

1. **Eksplicit Kontrol**: Du bruger måske AI'en som en *Diskret Komponent* eller i et foruddefineret workflow, der nødvendiggør udførelsen af en bestemt funktion på et bestemt tidspunkt. Ved at tvinge kaldet kan du garantere, at den ønskede funktion bliver aktiveret i stedet for at skulle bede AI'en pænt om at gøre det.
2. **Fejlfinding og Test**: Når man udvikler og tester AI-drevne applikationer, er muligheden for at tvinge funktionskald uvurderlig til fejlfindingsformål. Ved eksplicit at udløse specifikke funktioner kan du isolere og teste individuelle komponenter i din applikation. Dette giver dig mulighed for at verificere korrektheden af funktionsimplementeringerne, validere inputparametrene og sikre, at de forventede resultater returneres.

3. **Håndtering af Særtilfælde**: Der kan være særtilfælde eller exceptionelle scenarier, hvor AI'ens dynamiske udvælgelsesproces måske ikke vælger at udføre en funktion, som den burde, og du ved det baseret på eksterne processer. I sådanne tilfælde giver muligheden for at tvinge et funktionskald dig mulighed for at håndtere disse situationer eksplicit. Definer regler eller betingelser i din applikationslogik for at bestemme, hvornår AI'ens skøn skal tilsidesættes.
4. **Konsistens og Reproducerbarhed**: Hvis du har en specifik sekvens af funktioner, der skal udføres i en bestemt rækkefølge, garanterer tvungne kald, at den samme sekvens følges hver gang. Dette er særligt vigtigt i applikationer, hvor konsistens og forudsigelig adfærd er kritisk, såsom i finansielle systemer eller videnskabelige simuleringer.
5. **Ydelseoptimering**: I nogle tilfælde kan tvungne funktionskald føre til ydelseoptimeringer. Hvis du ved, at en specifik funktion er påkrævet til en bestemt opgave, og at AI'ens dynamiske udvælgelsesproces måske introducerer unødvendig overhead, kan du omgå udvælgelsesprocessen og direkte aktivere den påkrævede funktion. Dette kan hjælpe med at reducere latenstid og forbedre den overordnede effektivitet af din applikation.

Kort sagt giver muligheden for at tvinge funktionskald i AI-drevne applikationer eksplicit kontrol, hjælper med fejlfinding og test, håndterer særtilfælde og sikrer konsistens og reproducerbarhed. Det er et kraftfuldt værktøj i dit arsenal, men vi bliver nødt til at diskutere endnu et aspekt af denne vigtige funktion.

I mange beslutningstagningsscenarier ønsker vi altid, at modellen foretager et funktionskald og måske aldrig ønsker, at modellen svarer kun med sin interne viden. For eksempel, hvis du router mellem flere modeller, der er specialiseret i forskellige opgaver (flersproget input, matematik osv.), kan du bruge den funktionskaldende model til at delegere forespørgsler til en af hjælpemodellerne og aldrig svare selvstændigt.

### Værktøjsvalgparameter

GPT-4 og andre sprogmodeller, der understøtter funktionskald, giver dig en `tool_choice`-parameter til at kontrollere, om værktøjsbrug er påkrævet som en del af en færdiggørelse. Denne parameter har tre mulige værdier:

- `auto` giver AI'en fuld frihed til at bruge et værktøj eller blot svare
- `required` fortæller AI'en, at den *skal* kalde et værktøj *i stedet for* at svare, men overlader valget af værktøjet til AI'en
- Den tredje mulighed er at indstille parameteren til `name_of_function`, som du ønsker at tvinge. Mere om det i næste afsnit.

Bemærk, at hvis du sætter tool choice til `required`, vil modellen blive tvunget til at vælge den mest relevante funktion at kalde blandt de tilgængelige funktioner, selv hvis ingen af dem rigtig passer til prompten. På udgivelsestidspunktet kender jeg ikke til nogen model, der vil returnere et tomt `tool_calls` svar eller på anden måde lade dig vide, at den ikke fandt en passende funktion at kalde.

## Tvungen Funktionskald for Struktureret Output

Muligheden for at tvinge et funktionskald giver dig en måde at fremtvinge strukturerede data fra en chat-færdiggørelse i stedet for selv at skulle udtrække det fra dens klartekstsvar.

Hvorfor er det en stor sag at tvinge funktioner til at få struktureret output? Kort sagt, fordi udtrækning af strukturerede data fra LLM-output er en pine i nakken. Du kan gøre dit liv lidt lettere ved at bede om data i XML, men så skal du parse XML. Og hvad gør du, når den XML mangler, fordi din AI svarede: "Jeg beklager, men jeg kan

> ikke generere de data, du har anmodet om, fordi bla, bla, bla...”

Når du bruger værktøjer på denne måde:

- Du bør sandsynligvis definere et enkelt værktøj i din anmodning
- Husk at tvinge brugen af dens funktion ved hjælp af `tool_choice`-parameteren
- Husk, at modellen vil videregive inputtet til værktøjet, så navnet på værktøjet og beskrivelsen skal være fra modellens perspektiv, ikke dit.

Dette sidste punkt fortjener et eksempel for klarhedens skyld. Lad os sige, at du beder AI'en om at lave sentimentanalyse på brugertekst. Funktionens navn ville ikke være `analyze_sentiment`, men snarere noget som `save_sentiment_analysis`. Det er AI'en, der laver sentimentanalysen, *ikke værktøjet*. Alt hvad værktøjet gør (set fra AI'ens perspektiv) er at gemme resultaterne af analysen.

Her er et eksempel på brug af Claude 3 til at optage et resumé af et billede i velstruktureret JSON, denne gang fra kommandolinjen ved hjælp af `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 givne eksempel bruger vi Claude 3-modellen fra Anthropic til at generere en struktureret JSON-oversigt af et billede. Sådan fungerer det:

1. Vi definerer et enkelt værktøj ved navn `record_summary` i request-payloadens `tools`-array. Dette værktøj er ansvarligt for at registrere en oversigt over billedet i velstruktureret JSON.
2. `record_summary`-værktøjet har et `input_schema`, der specificerer den forventede struktur for JSON-outputtet. Det definerer tre egenskaber:

    - `key_colors`: Et array af objekter, der repræsenterer nøglefarverne i billedet. Hvert farveobjekt har egenskaber for rød-, grøn- og blå-værdier (fra 0.0 til 1.0) og et menneskelæsbart farvenavn i snake_case-format.
    - `description`: En string-egenskab til en kort beskrivelse af billedet, begrænset til 1-2 sætninger.

    - `estimated_year`: En valgfri integer-egenskab for det estimerede år, billedet blev taget, hvis det ser ud til at være et ikke-fiktivt foto.

3. I `messages`-arrayet leverer vi billeddata som en base64-kodet streng sammen med mediatypen. Dette gør det muligt for modellen at behandle billedet som en del af inputtet.
4. Vi beder også Claude om at bruge `record_summary`-værktøjet til at beskrive billedet.
5. Når anmodningen sendes til Claude 3-modellen, analyserer den billedet og genererer en JSON-oversigt baseret på det specificerede `input_schema`. Modellen udtrækker nøglefarverne, giver en kort beskrivelse og estimerer året, billedet blev taget (hvis relevant).
6. Den genererede JSON-oversigt sendes som parametre til `record_summary`-værktøjet og giver en struktureret repræsentation af billedets væsentlige karakteristika.

Ved at bruge `record_summary`-værktøjet med et veldefineret `input_schema` kan vi opnå en struktureret JSON-oversigt af et billede uden at være afhængige af almindelig tekstudtrækning. Denne tilgang sikrer, at outputtet følger et konsistent format og nemt kan analyseres og behandles af efterfølgende komponenter i applikationen.

Evnen til at fremtvinge et funktionskald og specificere den forventede output-struktur er en kraftfuld funktion ved værktøjsbrug i AI-drevne applikationer. Det giver udviklere mere kontrol over det genererede output og forenkler integrationen af AI-genereret data i applikationens arbejdsgang.

## Udførelse af funktion(er)

Du har defineret funktioner og promptet din AI, som besluttede, at den skulle kalde en af dine funktioner. Nu er det tid for din applikationskode eller dit bibliotek, hvis du bruger

en Ruby gem som `raix-rails`, til at sende funktionskaldet og dets parametre til den tilsvarende implementering *i din applikationskode*.

Din applikationskode bestemmer, hvad der skal gøres med resultaterne af funktionsudførelsen. Måske involverer det, der skal gøres, en enkelt linje kode i en lambda, eller måske involverer det at kalde et eksternt API. Måske involverer det at kalde en anden AI-komponent, eller måske involverer det hundredvis eller endda tusindvis af kodelinjer i resten af dit system. Det er helt op til dig.

Nogle gange er funktionskaldet slutningen på operationen, men hvis resultaterne repræsenterer information i en tankerække, der skal fortsættes af AI'en, så skal din applikationskode indsætte udførelsesresultaterne i chat-transkriptet og lade AI'en fortsætte behandlingen.

For eksempel, her er en Raix-funktionserklæring brugt af Olympias `AccountManager` til at kommunikere med vores klienter som en del af en Intelligent Arbejdsgangsorkestration 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 måske ikke umiddelbart klart, hvad der sker her, så lad mig bryde det ned.

1. `AccountManager`-klassen definerer mange funktioner relateret til

kontohåndtering. Den kan ændre din plan, tilføje og fjerne teammedlemmer blandt andre ting.

2. Dens instruktioner på øverste niveau fortæller `AccountManager`, at den skal underrette kontoejeren om resultaterne af kontoanmodningen ved hjælp af funktionen `notify_account_owner`.
3. Den koncise definition af funktionen inkluderer dens:

- navn
- beskrivelse
- parametre `message: { type: "string" }`
- en blok der skal udføres, når funktionen kaldes

Efter at have opdateret transskriptionen med resultaterne af funktionsblokken, kaldes `chat_completion`-metoden igen. Denne metode er ansvarlig for at sende den opdaterede samtaletranskription tilbage til AI-modellen til videre behandling. Vi henviser til denne proces som en *samtalesløjfe*.

Når AI-modellen modtager en ny chatfuldførelsesanmodning med en opdateret transskription, har den adgang til resultaterne af den tidligere udførte funktion. Den kan analysere disse resultater, inkorporere dem i sin beslutningsproces og generere det næste svar eller handling baseret på samtalens samlede kontekst. Den kan vælge at udføre yderligere funktioner baseret på den opdaterede kontekst, eller den kan generere et endeligt svar på den oprindelige prompt, hvis den vurderer, at ingen yderligere funktionskald er nødvendige.

## Valgfri fortsættelse af den oprindelige prompt

Når du sender værktøjsresultaterne tilbage til LLM'en og fortsætter behandlingen af den oprindelige prompt, bruger AI'en disse resultater til enten at kalde yderligere funktioner eller generere et endeligt tekstsvar.

Nogle modeller som Coheres Command-R kan citere de specifikke værktøjer, de brugte i deres svar, hvilket giver yderligere gennemsigtighed og sporbarhed.

Afhængigt af den anvendte model vil resultaterne af funktionskaldet leve i transskriptionsmeddelelser, der har deres egen særlige rolle, eller blive afspejlet i en anden syntaks. Men den vigtige del er, at disse data er i transskriptionen, så AI'en kan tage dem i betragtning, når den beslutter, hvad der skal gøres næst.

En almindelig (og potentielt dyr) fejltilstand er at glemme at tilføje funktionsresultaterne til transskriptionen, før man fortsætter chatten. Som resultat vil AI'en blive promptet på stort set samme måde, som den blev, før den kaldte funktionen første gang. Med andre ord, så vidt AI'en ved, har den ikke kaldt funktionen endnu. Så den kalder den igen. Og igen. Og igen, for evigt indtil du afbryder den. Håber din kontekst ikke var for stor, og din model ikke var for dyr!

# Bedste praksis for værktøjsbrug

For at få mest muligt ud af værktøjsbrug, overvej følgende bedste praksis.

## Beskrivende definitioner

Giv klare og beskrivende navne og beskrivelser for hvert værktøj og dets inputparametre. Dette hjælper LLM'en med bedre at forstå formålet og mulighederne for hvert værktøj.

> Jeg kan fortælle dig fra erfaring, at den almindelige visdom der siger, at "navngivning er svært" gælder her; jeg har set dramatisk forskellige resultater fra LLM'er bare ved at ændre navnene på funktioner eller ordlyden af beskrivelser. Nogle gange forbedrer fjernelse af beskrivelser faktisk ydeevnen.

## Behandling af værktøjsresultater

Når du sender værktøjsresultater tilbage til LLM'en, skal du sikre, at de er velstrukturerede og omfattende. Brug meningsfulde nøgler og værdier til at repræsentere outputtet fra hvert værktøj. Eksperimenter med forskellige formater og se hvilke der virker bedst, fra JSON til almindelig tekst.

Resultatfortolkeren adresserer denne udfordring ved at anvende AI til at analysere resultaterne og give menneskevenlige forklaringer, sammenfatninger eller hovedpointer.

## Fejlhåndtering

Implementer robuste fejlhåndteringsmekanismer til at håndtere tilfælde, hvor LLM'en kan generere ugyldige eller ikke-understøttede inputparametre for værktøjskald. Håndter og genopret elegant fra eventuelle fejl, der kan opstå under værktøjsudførelse.

En særdeles god egenskab ved AI'en er, at den forstår fejlmeddelelser! Hvilket betyder, at hvis du arbejder i en hurtig og beskidt tankegang, kan du simpelthen fange eventuelle undtagelser genereret i implementeringen af et værktøj og sende det tilbage til AI'en, så den ved, hvad der skete!

For eksempel, her er en forenklet version af implementeringen af Google-søgning i Olympia:

```
  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-søgninger i Olympia er en totrinsproces. Først udfører du søgningen, derefter opsummerer du resultaterne. Hvis der opstår en fejl, uanset hvad det er, bliver fejlmeddelelsen pakket sammen og sendt tilbage til AI'en. Denne teknik er fundamentet for praktisk talt alle *Intelligent Fejlhåndterings*-mønstre

Lad os for eksempel sige, at `GoogleSearch` API-kaldet fejler på grund af en 503 Servicen er ikke tilgængelig-undtagelse. Det bobler op til redningen på øverste niveau, og beskrivelsen af fejlen sendes tilbage til AI'en som resultatet af funktionskaldet. I stedet for bare at give brugeren en blank skærm eller teknisk fejl, siger AI'en noget i retning af "Jeg beklager, men jeg kan ikke få adgang til mine Google-søgefunktioner i øjeblikket. Jeg kan prøve igen senere, hvis du ønsker det."

Dette kan måske virke som et smart trick, men overvej en anden type fejl, hvor AI'en kaldte et eksternt API og havde direkte kontrol over de parametre, der skulle sendes til API'et. Måske lavede den en fejl i måden, den genererede disse parametre på? Forudsat at fejlmeddelelsen fra det eksterne API er detaljeret nok, betyder det at sende fejlmeddelelsen tilbage til den kaldende AI, at den kan genoverveje disse parametre og prøve igen. Automatisk. Uanset hvad fejlen var.

Tænk nu på, hvad det ville kræve at genskabe den slags robust fejlhåndtering i *normal* kode. Det er praktisk talt umuligt.

## Iterativ Forbedring

Hvis LLM'en ikke anbefaler de passende værktøjer eller genererer suboptimale svar, skal du iterere på værktøjsdefinitionerne, beskrivelserne og inputparametrene. Fortsæt med at forfine og forbedre værktøjsopsætningen baseret på den observerede adfærd og ønskede resultater.

1. Start med simple værktøjsdefinitioner: Begynd med at definere værktøjer med klare og koncise navne, beskrivelser og inputparametre. Undgå at overkomplicere værktøjsopsætningen i starten og fokuser på kernefunktionaliteten. Hvis du for eksempel ønsker at gemme resultaterne af sentimentanalyse, start med en grundlæggende definition 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 værktøjsdefinitioner på plads, test dem med forskellige prompts og observer, hvordan LLM'et interagerer med værktøjet. Vær opmærksom på kvaliteten og relevansen af de genererede svar. Hvis LLM'et genererer suboptimale svar, er det tid til at forfine værktøjsdefinitionerne.
3. Forfin beskrivelser: Hvis LLM'et misforstår formålet med et værktøj, så prøv at forfine værktøjets beskrivelse. Tilføj mere kontekst, eksempler eller præciseringer

for at guide LLM'et i at bruge værktøjet effektivt. For eksempel kan du opdatere beskrivelsen af sentimentanalyseværktøjet til mere specifikt at adressere den *emotionelle tone* i den tekst, der 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 inputparametre: Hvis LLM'en genererer ugyldige eller irrelevante inputparametre til et værktøj, bør du overveje at justere parameterdefinitionerne. Tilføj mere specifikke begrænsninger, valideringsregler eller eksempler for at tydeliggøre det forventede inputformat.
5. Iterer baseret på feedback: Overvåg løbende dine værktøjers ydeevne og indsaml feedback fra brugere eller interessenter. Brug denne feedback til at identificere områder, der kan forbedres, og foretag løbende forbedringer af værktøjsdefinitionerne. Hvis brugerne for eksempel rapporterer, at analysen ikke håndterer sarkasme særlig godt, kan du tilføje en bemærkning 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 iterativt at forfine dine værktøjsdefinitioner baseret på observeret adfærd og feedback kan du gradvist forbedre ydeevnen og effektiviteten af din AI-drevne applikation. Husk at holde værktøjsdefinitionerne klare, præcise og fokuserede på den specifikke opgave. Test og valider regelmæssigt værktøjsinteraktionerne for at sikre, at de stemmer overens med dine ønskede resultater.

# Sammensætning og Kædekobling af Værktøjer

Et af de mest kraftfulde aspekter ved værktøjsbrug, som kun er blevet antydet indtil nu, er muligheden for at sammensætte og kæde flere værktøjer sammen for at udføre komplekse opgaver. Ved omhyggeligt at designe dine værktøjsdefinitioner og deres input-/outputformater kan du skabe genbrugelige byggeklodser, der kan kombineres på forskellige måder.

Lad os se på et eksempel, hvor du bygger en dataanalysepipeline til din AI-drevne applikation. Du kan have følgende værktøjer:

1. **`DataRetrieval`**: Et værktøj, der henter data fra en database eller API baseret på specificerede kriterier.
2. **`DataProcessing`**: Et værktøj, der udfører beregninger, transformationer eller aggregeringer på de hentede data.
3. **`DataVisualization`**: Et værktøj, der præsenterer de behandlede data i et brugervenligt format, såsom diagrammer eller grafer.

Ved at kæde disse værktøjer sammen kan du skabe en kraftfuld arbejdsgang, der henter relevante data, behandler dem og præsenterer resultaterne på en meningsfuld måde. Her er hvordan værktøjsbrugens arbejdsgang kunne se ud:

1. LLM'en modtager en brugerforespørgsel, der beder om indsigt i salgsdata for en specifik produktkategori.
2. LLM'en vælger `DataRetrieval`-værktøjet og genererer de passende inputparametre for at hente de relevante salgsdata fra databasen.
3. De hentede data "videregives" til `DataProcessing`-værktøjet, som beregner målinger såsom samlet omsætning, gennemsnitlig salgspris og vækstrate.
4. De behandlede data bliver derefter bearbejdet af `DataVisualization`-værktøjet, som skaber et visuelt tiltalende diagram eller graf til at repræsentere indsigterne, og sender URL'en til diagrammet tilbage til LLM'en.

5. Endelig genererer LLM'en et formateret svar på brugerforespørgslen ved hjælp af markdown, der inkorporerer de visualiserede data og giver et sammendrag af de vigtigste resultater.

Ved at sammensætte disse værktøjer kan du skabe en problemfri dataanalysearbejdsgang, der nemt kan integreres i din applikation. Det smukke ved denne tilgang er, at hvert værktøj kan udvikles og testes uafhængigt og derefter kombineres på forskellige måder for at løse forskellige problemer.

For at muliggøre en gnidningsløs sammensætning og kædekobling af værktøjer er det vigtigt at definere klare input- og outputformater for hvert værktøj.

For eksempel kunne `DataRetrieval`-værktøjet acceptere parametre såsom databaseforbindelsesdetaljer, tabelnavn og forespørgselsbetingelser og returnere resultatsættet som et struktureret JSON-objekt. `DataProcessing`-værktøjet kan så forvente dette JSON-objekt som input og producere et transformeret JSON-objekt som output. Ved at standardisere dataflowet mellem værktøjer kan du sikre kompatibilitet og genbrugelighed.

Når du designer dit værktøjsøkosystem, så tænk over hvordan forskellige værktøjer kan kombineres for at adressere almindelige anvendelsestilfælde i din applikation. Overvej at skabe højniveauværktøjer, der indkapsler almindelige arbejdsgange eller forretningslogik, hvilket gør det lettere for LLM'en at vælge og bruge dem effektivt.

Husk, at styrken ved værktøjsbrug ligger i den fleksibilitet og modularitet, det giver. Ved at nedbryde komplekse opgaver i mindre, genbrugelige værktøjer kan du skabe en robust og tilpasningsdygtig AI-dreven applikation, der kan tackle en bred vifte af udfordringer.

# Fremtidige Retninger

Efterhånden som området for AI-dreven applikationsudvikling udvikler sig, kan vi forvente yderligere fremskridt i værktøjsbrugsfunktionaliteter. Nogle potentielle fremtidige retninger omfatter:

1. **Multi-hop Værktøjsbrug**: LLM'er kan muligvis beslutte, hvor mange gange de skal bruge værktøjer for at generere et tilfredsstillende svar. Dette kunne involvere flere runder af værktøjsvalg og -udførelse baseret på mellemliggende resultater.
2. **Foruddefinerede Værktøjer**: AI-platforme kan muligvis tilbyde et sæt foruddefinerede værktøjer, som udviklere kan udnytte uden videre tilpasning, såsom Python-fortolkere, websøgningsværktøjer eller almindelige hjælpefunktioner.
3. **Problemfri Integration**: Efterhånden som værktøjsbrug bliver mere udbredt, kan vi forvente bedre integration mellem AI-platforme og populære udviklingsrammer, hvilket gør det lettere for udviklere at inkorporere værktøjsbrug i deres applikationer.

Værktøjsbrug er en kraftfuld teknik, der gør det muligt for udviklere at udnytte det fulde potentiale af LLM'er i AI-drevne applikationer. Ved at forbinde LLM'er til eksterne værktøjer og ressourcer kan du skabe mere dynamiske, intelligente og kontekstbevidste systemer, der kan tilpasse sig brugerens behov og levere værdifulde indsigter og handlinger.

Mens værktøjsbrug tilbyder enorme muligheder, er det vigtigt at være opmærksom på potentielle udfordringer og overvejelser. Et centralt aspekt er at håndtere kompleksiteten af værktøjsinteraktioner og sikre stabilitet og pålidelighed i det samlede system. Du skal håndtere scenarier, hvor værktøjskald kan fejle, returnere uventede resultater eller have konsekvenser for ydeevnen. Derudover bør du overveje sikkerheds- og adgangskontrolforanstaltninger for at forhindre uautoriseret eller ondsindet brug af værktøjer. Korrekt fejlhåndtering, logning og overvågningsmekanismer er afgørende for at opretholde integriteten og ydeevnen i din AI-drevne applikation.

Når du udforsker mulighederne for værktøjsbrug i dine egne projekter, så husk at begynde med klare målsætninger, design velstrukturerede værktøjsdefinitioner og

iterer på baggrund af feedback og resultater. Med den rigtige tilgang og tankegang kan værktøjsbrug låse op for nye niveauer af innovation og værdi i dine AI-drevne applikationer

# Strømbehandling

Streaming af data over HTTP, også kendt som server-sendte begivenheder (SSE), er en mekanisme, hvor serveren kontinuerligt sender data til klienten, efterhånden som de bliver tilgængelige, uden at klienten eksplicit skal anmode om det. Da AI'ens svar genereres trinvist, giver det mening at skabe en responsiv brugeroplevelse ved at vise AI'ens output, mens det bliver genereret. Og faktisk tilbyder alle AI-udbyder-API'er, som jeg kender til, streaming-svar som en mulighed i deres færdiggørelsesendpoints.

Grunden til, at dette kapitel optræder her i bogen, lige efter Brug af værktøjer, er på grund af, hvor kraftfuldt det kan være at kombinere brugen af værktøjer med live AI-svar til brugerne. Dette muliggør dynamiske og interaktive oplevelser, hvor AI'en kan behandle brugerinput, udnytte forskellige værktøjer og funktioner efter eget skøn og derefter give realtidssvar.

For at opnå denne problemfrie interaktion skal du skrive strømhåndterere, der kan ekspedere AI-aktiverede værktøjsfunktionskald såvel som almindelig tekstoutput til

slutbrugeren. Behovet for at løkke efter behandling af en værktøjsfunktion tilføjer en interessant udfordring til opgaven.

## Implementering af en ReplyStream

For at demonstrere hvordan strømbehandling kan implementeres, vil dette kapitel dykke dybt ned i en forenklet version af `ReplyStream`-klassen, der bruges i Olympia. Instanser af denne klasse kan sendes som `stream`-parameteren i AI-klientbiblioteker såsom ruby-openai og openrouter

Her er hvordan jeg bruger `ReplyStream` i Olympias `PromptSubscriber`, som lytter via Wisper efter oprettelsen af nye brugermeddelelser.

```
class PromptSubscriber
  include Raix::ChatCompletion
  include Raix::PromptDeclarations

  # many other declarations omitted...

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

  def message_created(message) # invoked by Wisper
    return unless message.role.user? && message.content?

    # rest of the implementation omitted...
```

Ud over en `context`-reference til den prompt-abonnent, der instantierede den, har `ReplyStream`-klassen også instansvariabler til at gemme en buffer med modtaget data samt arrays til at holde styr på funktionsnavne og argumenter, der bliver anvendt 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
```

`initialize`-metoden opsætter den initielle tilstand af `ReplyStream`-instansen ved at initialisere bufferen, konteksten og andre variabler.

`call`-metoden er hovedindgangspunktet for behandling af streaming-dataene. Den tager en `chunk` af data (repræsenteret som et hash) og en valgfri `bytesize`-parameter, som i vores eksempel ikke bliver brugt. Inde i denne metode bruger klassen mønstergenkendelse til at håndtere forskellige scenarier baseret på strukturen af den modtagne chunk.

At kalde `deep_symbolize_keys` på chunken hjælper med at gøre mønstergenkendelsen mere elegant ved at lade os arbejde med symboler i stedet for strenge.

```
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 mønster, vi matcher efter, er et værktøjskald sammen med dets tilhørende funktionsnavn. Hvis vi opdager et, lægger vi det i `f_name`-arrayet. Vi gemmer funktionsnavne i et indekseret array, fordi modellen er i stand til at udføre parallelle funktionskald, hvor den sender mere end én funktion til udførelse ad gangen.

> Parallel funktionskald er en AI-models evne til at udføre flere funktionskald sammen, hvilket tillader effekterne og resultaterne af disse funktionskald at blive løst parallelt. Dette er særligt nyttigt, hvis funktioner tager lang tid, og det reducerer antallet af forespørgsler til API'et, hvilket igen kan spare et betydeligt token-forbrug.

Dernæst skal vi matche argumenterne, der svarer til funktionskaldene.

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

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

Ligesom vi håndterede funktionsnavnene, gemmer vi argumenterne i et indekseret array.

Dernæst holder vi øje med normale brugervendte beskeder, som vil ankomme fra serveren én token ad gangen og blive tildelt `new_content`-variablen. Vi skal også holde øje med `finish_reason`. Den vil være `nil` indtil det sidste stykke af output-sekvensen.

```
  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
```

Vigtigt er det, at vi tilføjer et mønstergenkendelsesudtryk til at håndtere fejlmeddelelser sendt af AI-modeludbyderen. I lokale udviklingsmiljøer kaster vi en undtagelse, men i produktion logger vi fejlen og afslutter.

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

Den afsluttende else-sætning i case vil blive udført, hvis ingen af de foregående mønstre matchede. Det er blot en sikkerhedsforanstaltning, så vi opdager det, hvis AI-modellen begynder at sende os ukendte bidder.

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

Metoden `send_to_client` er ansvarlig for at sende det bufferede indhold til klienten. Den kontrollerer, at bufferen ikke er tom, opdaterer bot-beskedens indhold, renderer bot-beskeden og gemmer indholdet i databasen for at 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 kaldes, når strømbehandlingen er færdig. Den udfører funktionskaldene, hvis der er modtaget nogen under strømmen, opdaterer bot-beskeden med det endelige indhold og andre relevante oplysninger og nulstiller funktionskaldshistorikken

```
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 beslutter sig for at kalde en funktion, skal du "afsende" dette funktionskald (navn og argumenter) på en sådan måde, at det bliver udført, og `function_call` og `function_result` beskeder bliver tilføjet til samtalehistorikken

Ud fra min erfaring er det bedre at håndtere oprettelsen af funktionsbeskeder ét sted i din kodebase, i stedet for at være afhængig af værktøjsimplementeringerne. Det er ikke kun mere overskueligt, men har også en meget vigtig praktisk grund: hvis AI-modellen kalder en funktion, og ikke ser de resulterende kald og resultatbeskeder i historikken, når du looper, vil den *kalde den samme funktion igen*. Potentielt i det uendelige. Husk, at AI'en er fuldstændig tilstandsløs, så medmindre du sender disse funktionskald tilbage til den, er de aldrig sket.

```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
```

At rydde funktionskaldsoversigten efter afsendelse er lige så vigtigt som at sikre, at kaldet og resultaterne ender i dit transskript, så du ikke bare bliver ved med at kalde de samme funktioner igen og igen hver gang du løber gennem løjfen.

## "Samtalesløjfen"

I `PromptSubscriber`-klassen bruger vi `prompt`-metoden fra `PromptDeclarations`-modulet til at definere samtalesløjfens opførsel. `until`-parameteren er sat til `-> { bot_message.complete? }`, hvilket betyder, at løjfen vil fortsætte indtil `bot_message` er markeret som færdig.

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

Men hvornår markeres `bot_message` som fuldført? Hvis du har glemt det, så kig tilbage på linje 13 i `finalize`-metoden.

Lad os gennemgå hele strømbehandlingslogikken.

1. `PromptSubscriber` modtager en ny brugerbesked via `message_created`-metoden, som aktiveres af Wisper pub/sub-systemet, hver gang slutbrugeren opretter et nyt prompt.
2. Klassemetoden `prompt` definerer deklarativt chatfærdiggørelseslogikken for `PromptSubscriber`. AI-modellen vil udføre en chatfærdiggørelse med brugerens beskedindhold, en ny instans af `ReplyStream` som strømparameter og den specificerede løkkebetingelse.
3. AI-modellen behandler promptet og begynder at generere et svar. Efterhånden som svaret strømmes, kaldes `call`-metoden på `ReplyStream`-instansen for hver datadel.
4. Hvis AI-modellen beslutter at kalde en værktøjsfunktion, udtrækkes funktionsnavnet og argumenterne fra datadelen og gemmes henholdsvis i `f_name`- og `f_arguments`-arrayerne.
5. Hvis AI-modellen genererer brugervendt indhold, bliver det buffereret og sendt til klienten via `send_to_client`-metoden.
6. Når strømbehandlingen er færdig, kaldes `finalize`-metoden. Hvis der blev kaldt værktøjsfunktioner under strømmen, bliver de ekspederet ved hjælp af `dispatch`-metoden i `PromptSubscriber`.
7. `dispatch`-metoden tilføjer en `function_call`-besked til samtaleudskriften, udfører den tilsvarende værktøjsfunktion og tilføjer en `function_result`-besked til udskriften med resultatet af funktionskaldet.
8. Efter ekspedition af værktøjsfunktionerne ryddes funktionskaldshistorikken for at forhindre dublerede funktionskald i efterfølgende løkker.
9. Hvis der ikke blev kaldt nogen værktøjsfunktioner, opdaterer `finalize`-metoden `bot_message` med det endelige indhold, markerer det som fuldført og sender den opdaterede besked til klienten.
10. Løkkebetingelsen `-> { bot_message.complete? }` evalueres. Hvis `bot_message` ikke er markeret som fuldført, fortsætter løkken, og det oprindelige prompt indsendes igen med den opdaterede samtaleudskrift.

11. Trin 3-10 gentages, indtil `bot_message` er markeret som fuldført, hvilket indikerer, at AI-modellen har afsluttet genereringen af sit svar, og ingen yderligere værktøjsfunktioner skal udføres.

Ved at implementere denne samtaleløkke gør du det muligt for AI-modellen at indgå i en frem-og-tilbage-interaktion med applikationen, udføre værktøjsfunktioner efter behov og generere brugervendte svar, indtil samtalen når en naturlig afslutning.

Kombinationen af strømbehandling og samtaleløkken muliggør dynamiske og interaktive AI-drevne oplevelser, hvor AI-modellen kan behandle brugerinput, udnytte forskellige værktøjer og funktioner og give realtidssvar baseret på den udviklende samtalekontekst.

## Automatisk Fortsættelse

Det er vigtigt at være opmærksom på AI-outputbegrænsninger. De fleste modeller har et maksimalt antal tokens, de kan generere i et enkelt svar, hvilket bestemmes af `max_tokens`-parameteren. Hvis AI-modellen når denne grænse under generering af et svar, vil den brat stoppe og indikere, at outputtet blev afkortet.

I streamingsvaret fra AI-platformens API kan du opdage denne situation ved at undersøge `finish_reason`-variablen i datadelen. Hvis `finish_reason` er sat til `"length"` (eller en anden nøgleværdi specifik for modellen), betyder det, at modellen nåede sin maksimale token-grænse under genereringen, og outputtet er blevet afkortet.

En måde at håndtere dette scenarie elegant på og give en problemfri brugeroplevelse er at implementere en automatisk fortsættelsesmekanisme i din strømbehandlingslogik. Ved at tilføje et mønstermatch for længderelaterede afslutningsårsager kan du vælge at løkke og automatisk fortsætte outputtet fra hvor det slap.

Her er et bevidst forenklet eksempel på, hvordan du kan modificere `call`-metoden i `ReplyStream`-klassen for at understøtte automatisk fortsættelse:

```
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)
  conversation.user_message!("please continue", visible: false)
  bot_message.update_column(:created_at, Time.current)
end
```

I denne modificerede version, når `finish_reason` indikerer afkortet output, i stedet for at afslutte strømmen, tilføjer vi et par beskeder til transskriptet uden at afslutte det, flytter den oprindelige brugervendte svarbesked til "bunden" af transskriptet ved at opdatere dens `created_at`-attribut, og lader derefter løkken fortsætte, så AI'en fortsætter med at generere hvor den slap.

Husk at AI-fuldførelsesendepunktet er tilstandsløst. Det "kender" kun det, du fortæller

det via transskriptet. I dette tilfælde er måden, hvorpå vi kommunikerer til AI'en, at den blev afbrudt, ved at tilføje "usynlige" (for slutbrugeren) beskeder til transskriptet. Husk dog, at dette er et bevidst forenklet eksempel. En rigtig implementering ville have behov for yderligere transskripthåndtering for at sikre, at vi ikke spildte tokens og/eller forvirrede AI'en med duplikerede assistentbeskeder i transskriptet.

En rigtig implementering af auto-fortsættelse bør også have såkaldt "kredsløbsbryder-logik" på plads for at forhindre løbsk løkkekørsel. Årsagen er, at AI'en med bestemte typer af brugerprompts og lave `max_tokens`-indstillinger kunne fortsætte med at loop brugervendt output endeløst.

> Husk, at hver løkke kræver en separat forespørgsel, og at hver forespørgsel forbruger hele dit transskript igen. Du bør helt sikkert overveje afvejningerne mellem brugeroplevelse og API-forbrug, når du beslutter, om du vil implementere auto-fortsættelse i din applikation. Auto-fortsættelse kan især være farligt dyrt, særligt når der bruges premium kommercielle modeller.

## Konklusion

Strømbehandling er et kritisk aspekt af at bygge AI-drevne applikationer, der kombinerer værktøjsanvendelse med live AI-svar. Ved effektivt at håndtere streamingdata fra AI-platform-API'er kan du levere en problemfri og interaktiv brugeroplevelse, håndtere store svar, optimere ressourceforbruget og elegant håndtere fejl.

Den leverede `Conversation::ReplyStream`-klasse demonstrerer, hvordan strømbehandling kan implementeres i en Ruby-applikation ved hjælp af mønstergenkendelse og hændelsesdrevet arkitektur. Ved at forstå og udnytte

strømbehandlingsteknikker kan du frigøre det fulde potentiale af AI-integration i dine applikationer og levere kraftfulde og engagerende brugeroplevelser.

# Selvhelende data

Selvhelende data er en kraftfuld tilgang til at sikre dataintegritet, konsistens og kvalitet i applikationer ved at udnytte mulighederne i store sprogmodeller (LLMs). Denne kategori af mønstre fokuserer på idéen om at bruge AI til automatisk at opdage, diagnosticere og korrigere dataanomalier, inkonsistens eller fejl, og dermed reducere byrden for udviklere og opretholde et højt niveau af datapålidelighed.

I kernen anerkender de selvhelende datamønstre, at data er livsblodet i enhver applikation, og at sikring af deres nøjagtighed og integritet er afgørende for applikationens korrekte funktion og brugeroplevelse. Dog kan styring og vedligeholdelse af datakvalitet være en kompleks og tidskrævende opgave, især når applikationer vokser i størrelse og kompleksitet. Det er her, AI's kraft kommer i spil.

I de selvhelende datamønstre anvendes AI-workers til kontinuerligt at overvåge og analysere din applikations data. Disse modeller har evnen til at forstå og fortolke

mønstre, relationer og anomalier i dataene. Ved at udnytte deres evner inden for naturlig sprogbehandling og forståelse kan de identificere potentielle problemer eller inkonsistenser i dataene og træffe passende foranstaltninger for at rette dem.

Processen med selvhelende data involverer typisk flere centrale trin:

1. **Dataovervågning**: AI-workers overvåger konstant applikationens datastrømme, databaser eller lagringssystemer for at lede efter tegn på anomalier, inkonsistens eller fejl. Alternativt kan du aktivere en AI-komponent som reaktion på en undtagelse.
2. **Anomalidetektion**: Når et problem opdages, analyserer AI-workeren dataene i detaljer for at identificere problemets specifikke karakter og omfang. Dette kan omfatte opdagelse af manglende værdier, inkonsistente formater eller data, der overtræder foruddefinerede regler eller begrænsninger.
3. **Diagnose og korrektion**: Når problemet er identificeret, bruger AI-workeren sin viden og forståelse af datadomænet til at bestemme den passende handlingsplan. Dette kan involvere automatisk korrektion af data, udfyldning af manglende værdier eller markering af problemet til menneskelig intervention, hvis nødvendigt.
4. **Kontinuerlig læring (valgfrit, afhængigt af anvendelsestilfælde)**: Når din AI-worker møder og løser forskellige dataproblemer, kan den output beskrivelser af, hvad der skete, og hvordan den reagerede. Disse metadata kan fødes ind i læringsprocesser, der gør det muligt for dig (og måske den underliggende model via finjustering) at blive mere effektiv over tid i at identificere og løse dataanomalier.

Ved automatisk at opdage og korrigere dataproblemer kan du sikre, at din applikation opererer med data af høj kvalitet og pålidelighed. Dette reducerer risikoen for fejl, inkonsistens eller datarelaterede fejl, der påvirker applikationens funktionalitet eller brugeroplevelse.

Når du har AI-workers til at håndtere opgaven med dataovervågning og -korrektion, kan du fokusere dine kræfter på andre kritiske aspekter af applikationen. Dette sparer tid og ressourcer, der ellers ville blive brugt på manuel datarensning og vedligeholdelse. Faktisk bliver manuel håndtering af datakvalitet stadig mere udfordrende, efterhånden som dine applikationer vokser i størrelse og kompleksitet. "Selvhelende data"-mønstrene skalerer effektivt ved at udnytte AI's kraft til at håndtere store mængder data og opdage problemer i realtid.

På grund af deres natur kan AI-modeller tilpasse sig ændrede datamønstre, skemaer eller krav over tid med lille eller ingen overvågning. Så længe deres direktiver giver tilstrækkelig vejledning, især vedrørende tilsigtede resultater, kan din applikation muligvis udvikle sig og håndtere nye datascenarier uden at kræve omfattende manuel intervention eller kodeændringer.

De selvhelende datamønstre harmonerer godt med de andre kategorier af mønstre, vi har diskuteret, såsom "Multitude of Workers". Selvhelende datafunktionalitet kan ses som en specialiseret type worker, der specifikt fokuserer på at sikre datakvalitet og -integritet. Denne type worker fungerer sammen med andre AI-workers, hvor hver bidrager til forskellige aspekter af applikationens funktionalitet.

Implementering af selvhelende datamønstre i praksis kræver omhyggelig design og integration af AI-modeller i applikationsarkitekturen. På grund af risikoen for datatab og -korruption bør du definere klare retningslinjer for, hvordan du vil bruge denne teknik. Du bør også overveje faktorer som ydeevne, skalerbarhed og datasikkerhed.

## Praktisk casestudie: Reparation af ødelagt JSON

En af de mest praktiske og bekvemme måder at udnytte selvhelende data på er også meget simpel at forklare: reparation af ødelagt JSON.

Denne teknik kan anvendes på den almindelige udfordring med at håndtere ufuldstændige eller inkonsistente data genereret af LLMs, såsom ødelagt JSON, og giver en tilgang til automatisk at opdage og korrigere disse problemer.

Hos Olympia støder jeg jævnligt på scenarier, hvor LLM'er genererer JSON-data, som ikke er fuldstændig valide. Dette kan ske af forskellige årsager, såsom at LLM'en tilføjer kommentarer før eller efter selve JSON-koden, eller introducerer syntaksfejl som manglende kommaer eller ikke-escapede dobbelte citationstegn. Disse problemer kan føre til parsing-fejl og forårsage forstyrrelser i applikationens funktionalitet.

For at løse dette problem har jeg implementeret en praktisk løsning i form af en JsonFixer-klasse. Denne klasse implementerer "Selvhelbredende Data"-mønsteret ved at tage den ødelagte JSON som input og udnytte en LLM til at reparere den, mens den bevarer så meget information og intention som muligt.

```
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
```

Bemærk hvordan `JsonFixer` bruger Ventriloquist til at styre AI'ens svar.

Processen med selvhelbredende JSON-data fungerer som følger:

1. **JSON-generering**: En LLM bruges til at generere JSON-data baseret på bestemte prompts eller krav. På grund af LLM'ers natur vil den genererede JSON dog ikke altid være perfekt gyldig. JSON-parseren vil naturligvis udløse en `ParserError`, hvis du giver 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
```

Bemærk, at fejlmeddelelsen også sendes til `JSONFixer`-kaldet, så den ikke behøver at antage fuldt ud, hvad der er galt med dataene, især da parseren ofte vil fortælle dig præcis, hvad der er galt.

2. **LLM-baseret Korrektion**: `JSONFixer`-klassen sender den ødelagte JSON tilbage til en LLM sammen med en specifik prompt eller instruktion om at rette JSON'en, mens den originale information og hensigt bevares så meget som muligt. LLM'en, som er trænet på store mængder data og har en forståelse af JSON-syntaks, forsøger at rette fejlene og generere en gyldig JSON-streng. Response Fencing bruges til at begrænse LLM'ens output, og vi vælger Mixtral 8x7B som AI-modellen, da den er særligt god til denne type opgave.

3. **Validering og Integration**: Den rettede JSON-streng, der returneres af LLM'en, bliver parset af selve `JSONFixer`-klassen, fordi den kaldte `chat_completion(json: true)`. Hvis den rettede JSON består valideringen, integreres den tilbage i applikationens arbejdsgang, hvilket gør det muligt for applikationen at fortsætte databehandlingen uden problemer. Den dårlige JSON er blevet "helbredt".

Selvom jeg har skrevet og omskrevet min egen `JSONFixer`-implementering adskillige gange, tvivler jeg på, at den samlede tid investeret i alle disse versioner er mere end en time eller to.

Bemærk, at bevarelse af hensigt er et nøgleelement i ethvert selvhelende data-mønster. Den LLM-baserede korrektionsproces sigter mod at bevare den originale information og hensigt i den genererede JSON så meget som muligt. Dette sikrer, at den rettede JSON bevarer sin semantiske betydning og kan bruges effektivt inden for applikationens kontekst.

Denne praktiske implementering af "Selvhelende Data"-tilgangen i Olympia demonstrerer tydeligt, hvordan AI, specifikt LLM'er, kan udnyttes til at løse virkelige dataudfordringer. Det viser styrken ved at kombinere traditionelle programmeringsteknikker med AI-kapaciteter for at bygge robuste og effektive applikationer.

## Postels Lov og "Selvhelende Data"-Mønsteret

"Selvhelende Data", som eksemplificeret ved JSONFixer-klassen, stemmer godt overens med princippet kendt som Postels Lov, også kendt som Robusthedsprincippet. Postels Lov siger:

"Vær konservativ i det, du gør, vær liberal i det, du accepterer fra andre."

Dette princip, oprindeligt formuleret af Jon Postel, en pioner inden for det tidlige internet, understreger vigtigheden af at bygge systemer, der er tolerante over for forskellige eller endda let ukorrekte input, mens de opretholder streng overholdelse af specificerede protokoller ved afsendelse af output.

I konteksten af "Selvhelende Data" legemliggør JSONFixer-klassen Postels Lov ved at være liberal i accepten af ødelagt eller ufuldkommen JSON-data genereret af LLM'er. Den afviser eller fejler ikke øjeblikkeligt, når den møder JSON, der ikke strengt overholder det forventede format. I stedet tager den en tolerant tilgang og forsøger at rette JSON'en ved hjælp af LLM'ernes kraft.

Ved at være liberal i accepten af ufuldkommen JSON demonstrerer JSONFixer-klassen robusthed og fleksibilitet. Den anerkender, at data i den virkelige verden ofte kommer i forskellige former og ikke altid overholder strenge specifikationer. Ved elegant at håndtere og korrigere disse afvigelser sikrer klassen, at applikationen kan fortsætte med at fungere problemfrit, selv når der er ufuldkomne data.

På den anden side overholder JSONFixer-klassen også det konservative aspekt af Postels Lov, når det kommer til output. Efter at have rettet JSON'en ved hjælp af LLM'er, validerer klassen den korrigerede JSON for at sikre, at den strengt overholder det forventede format. Den opretholder dataenes integritet og korrekthed, før de sendes videre til andre dele af applikationen. Denne konservative tilgang garanterer, at outputtet fra JSONFixer-klassen er pålideligt og konsistent, hvilket fremmer interoperabilitet og forhindrer spredning af fejl.

Interessante fakta om Jon Postel:

- Jon Postel (1943-1998) var en amerikansk datalog, som spillede en afgørende rolle i udviklingen af internettet. Han var kendt som "Internettets Gud" for hans betydelige bidrag til de underliggende protokoller og standarder.
- Postel var redaktør for Request for Comments (RFC) dokumentserien, som er en serie af tekniske og organisatoriske noter om internettet. Han forfattede eller medforfattede over 200 RFC'er, inklusive de grundlæggende protokoller

såsom TCP, IP og SMTP.
- Ud over hans tekniske bidrag var Postel kendt for sin ydmyge og samarbejdende tilgang. Han troede på vigtigheden af at nå konsensus og arbejde sammen om at bygge et robust og interoperabelt netværk.
- Postel fungerede som direktør for Computer Networks Division ved Information Sciences Institute (ISI) ved University of Southern California (USC) fra 1977 indtil hans alt for tidlige død i 1998.
- Som anerkendelse for hans enorme bidrag blev Postel posthumt tildelt den prestigefyldte Turing Award i 1998, ofte omtalt som "Datalogiens Nobelpris."

`JSONFixer`-klassen fremmer robusthed, fleksibilitet og interoperabilitet, hvilket var kerneværdier, som Postel opretholdt gennem hele sin karriere. Ved at bygge systemer, der er tolerante over for ufuldkommenheder, mens de samtidig opretholder streng overholdelse af protokoller, kan vi skabe applikationer, der er mere modstandsdygtige og tilpasningsdygtige over for virkelighedens udfordringer.

## Overvejelser og Kontraindikationer

Anvendeligheden af selvhelbredende datatilgange afhænger fuldstændigt af, hvilken type data din applikation håndterer. Der er en grund til, at du måske ikke ønsker at monkeypatch `JSON.parse` til automatisk at selvkorrigere *alle JSON-parsing fejl* i din applikation: ikke alle fejl kan eller bør korrigeres automatisk.

Selvhelbredende er særligt problematisk, når det er koblet sammen med lovmæssige eller compliance-krav relateret til datahåndtering og -behandling. Nogle brancher, såsom sundhedsvæsenet og finanssektoren, har så strenge regler vedrørende dataintegritet og sporbarhed, at enhver form for "black box" datakorrektion uden ordentligt tilsyn eller logføring kan overtræde disse regler. Det er afgørende at sikre, at alle selvhelbredende datateknikker, du udvikler, er i overensstemmelse med de gældende juridiske og

regulatoriske rammer.

Anvendelsen af selvhelbredende datateknikker, særligt dem der involverer AI-modeller, kan også have stor indvirkning på applikationens ydeevne og ressourceudnyttelse. Behandling af store mængder data gennem AI-modeller til fejldetektion og -korrektion kan være beregningsmæssigt krævende. Det er vigtigt at vurdere afvejningerne mellem fordelene ved selvhelbredende data og de tilhørende omkostninger i forhold til ydeevne og ressourcer.

Lad os dykke ned i de faktorer, der er involveret i at beslutte hvornår og hvor denne kraftfulde tilgang skal anvendes.

## Data Kritikalitet

Når man overvejer anvendelsen af selvhelbredende datateknikker, er det afgørende at vurdere kritikaliteten af de data, der behandles. Kritikalitetsniveauet henviser til vigtigheden og følsomheden af dataene i konteksten af din applikation og dens forretningsområde.

I nogle tilfælde er det måske ikke hensigtsmæssigt at korrigere datafejl automatisk, især hvis dataene er meget følsomme eller har juridiske implikationer. Overvej for eksempel følgende scenarier:

1. **Finansielle Transaktioner**: I finansielle applikationer, såsom banksystemer eller handelsplatforme, er datanøjagtighed af største betydning. Selv mindre fejl i finansielle data kan have betydelige konsekvenser, såsom forkerte kontosaldi, fejldirigerede midler eller fejlagtige handelsbeslutninger. I disse tilfælde kan automatiserede korrektioner uden grundig verifikation og revision medføre uacceptable risici.
2. **Medicinske Journaler**: Sundhedsapplikationer håndterer meget følsomme og fortrolige patientdata. Unøjagtigheder i medicinske journaler kan have alvorlige konsekvenser for patientsikkerheden og behandlingsbeslutninger. Automatisk

ændring af medicinske data uden ordentligt tilsyn og validering af kvalificeret sundhedspersonale kan overtræde lovkrav og bringe patientens velbefindende i fare.

3. **Juridiske Dokumenter**: Applikationer, der håndterer juridiske dokumenter, såsom kontrakter, aftaler eller retsdokumenter, kræver streng nøjagtighed og integritet. Selv mindre fejl i juridiske data kan have betydelige juridiske konsekvenser. Automatiserede korrektioner på dette område er måske ikke hensigtsmæssige, da dataene ofte kræver manuel gennemgang og verifikation af juridiske eksperter for at sikre deres gyldighed og retskraft.

I disse kritiske datascenarier opvejer risiciene forbundet med automatiserede korrektioner ofte de potentielle fordele. Konsekvenserne af at introducere fejl eller ændre data forkert kan være alvorlige og føre til økonomiske tab, juridiske forpligtelser eller endda skade på personer.

Når man håndterer meget kritiske data, er det essentielt at prioritere manuelle verifikations- og valideringsprocesser. Menneskeligt tilsyn og ekspertise er afgørende for at sikre dataenes nøjagtighed og integritet. Automatiserede selvhelbredende teknikker kan stadig anvendes til at markere potentielle fejl eller uoverensstemmelser, men den endelige beslutning om korrektioner bør involvere menneskelig bedømmelse og godkendelse.

Det er dog vigtigt at bemærke, at ikke alle data i en applikation nødvendigvis har samme kritikalitetsniveau. Inden for samme applikation kan der være delmængder af data, som er mindre følsomme eller har lavere konsekvenser, hvis der opstår fejl. I sådanne tilfælde kan selvhelbredende datateknikker anvendes selektivt på disse specifikke datadelmængder, mens kritiske data forbliver underlagt manuel verifikation.

Det afgørende er at vurdere kritikaliteten af hver datakategori i din applikation omhyggeligt og definere klare retningslinjer og processer for håndtering af korrektioner baseret på de tilknyttede risici og implikationer. Ved at skelne mellem kritiske (f.eks. hovedbøger, medicinske journaler) og ikke-kritiske data (f.eks. postadresser,

ressourceadvarsler), kan du finde en balance mellem at udnytte fordelene ved selvhelbredende datateknikker, hvor det er passende, og opretholde streng kontrol og tilsyn, hvor det er nødvendigt.

I sidste ende bør beslutningen om at anvende selvhelbredende datateknikker på kritiske data træffes i samråd med domæneeksperter, juridiske rådgivere og andre relevante interessenter. Det er essentielt at overveje de specifikke krav, regler og risici, der er forbundet med din applikations data og tilpasse datakorrekturstrategierne i overensstemmelse hermed.

## Fejlens Alvorlighed

Når man anvender selvhelbredende datateknikker, er det vigtigt at vurdere alvoren og påvirkningen af datafejlene. Ikke alle fejl er skabt lige, og den passende fremgangsmåde kan variere afhængigt af problemets alvorlighed.

Mindre uoverensstemmelser eller formateringsproblemer kan være egnede til automatisk korrektion. For eksempel kan en selvhelbredende dataarbejder, der er sat til at rette ødelagt JSON, håndtere manglende kommaer eller ikke-escapede anførselstegn uden at ændre væsentligt ved dataenes betydning eller struktur. Disse typer fejl er ofte lige til at rette og har minimal indvirkning på den overordnede dataintegritet.

Dog kan mere alvorlige fejl, der fundamentalt ændrer betydningen eller integriteten af dataene, kræve en anden tilgang. I sådanne tilfælde er automatiserede korrektioner måske ikke tilstrækkelige, og menneskelig indgriben kan være nødvendig for at sikre dataenes nøjagtighed og gyldighed.

Det er her, at konceptet med at bruge selve AI'en til at hjælpe med at bestemme fejlenes alvorlighed kommer i spil. Ved at udnytte AI-modellernes kapaciteter kan vi designe selvhelbredende databehandlere, der ikke kun korrigerer fejl, men også vurderer fejlenes alvorlighed og træffer velinformerede beslutninger om, hvordan de skal håndteres.

Lad os for eksempel se på en selvhelbredende databehandler med ansvar for at korrigere uoverensstemmelser i datastrømmen til en kundedatabase. Behandleren kan designes til

at analysere dataene og identificere potentielle fejl, såsom manglende eller modstridende information. I stedet for automatisk at korrigere alle fejl kan behandleren udstyres med yderligere værktøjskald, der gør det muligt at markere alvorlige fejl til menneskelig gennemgang.

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 eksempel er `CustomerDataHealer`-workeren designet til at identificere og korrigere uoverensstemmelser i kundedata. Igen bruger vi Response Fencing og Ventriloquist til at få struktureret output. Vigtigt er det, at workerens systemdirektiv indeholder instruktioner om at bruge `flag_for_review`-funktionen, hvis der opdages alvorlige fejl.

Når workeren behandler kundedataene, analyserer den dataene og forsøger at rette eventuelle uoverensstemmelser. Hvis workeren vurderer, at fejlene er alvorlige og kræver menneskelig indgriben, kan den bruge `flag_for_review`-værktøjet til at markere dataene og angive en årsag til markeringen.

`chat_completion`-metoden kaldes med `json: true` for at parse de korrigerede kundedata som JSON. Der er ingen mulighed for at loope efter et funktionskald, så resultatet vil være tomt, hvis `flag_for_review` blev aktiveret. Ellers opdateres kunden med de gennemgåede og potentielt korrigerede data.

Ved at inkorporere vurdering af fejlens alvorlighed og muligheden for at markere data til menneskelig gennemgang bliver den selvhelende dataworker mere intelligent og tilpasningsdygtig. Den kan håndtere mindre fejl automatisk, mens alvorlige fejl eskaleres til menneskeelige eksperter for manuel intervention.

De specifikke kriterier for at bestemme fejlens alvorlighed kan defineres i workerens direktiv baseret på domæneviden og forretningsmæssige krav. Faktorer som påvirkningen på dataintegritet, risikoen for datatab eller -korruption og konsekvenserne af ukorrekte data kan tages i betragtning ved vurdering af alvorsgraden.

Ved at udnytte AI til at vurdere fejlens alvorlighed og give muligheder for menneskelig intervention kan selvhelende datateknikker skabe balance mellem automatisering og opretholdelse af datanøjagtighed. Denne tilgang sikrer, at mindre fejl rettes effektivt, mens alvorlige fejl modtager den nødvendige opmærksomhed og ekspertise fra menneskelige bedømmere.

## Domænekompleksitet

Når man overvejer anvendelsen af selvhelende datateknikker, er det vigtigt at evaluere kompleksiteten af datademænet og de regler, der styrer dets struktur og relationer. Domænets kompleksitet kan have betydelig indflydelse på effektiviteten og gennemførligheden af automatiserede datakorrektionsmetoder.

Selvhelende datateknikker fungerer godt, når dataene følger veldefinerede mønstre og begrænsninger. I domæner hvor datastrukturen er relativt simpel, og relationerne mellem dataelementer er ligetil, kan automatiske korrektioner anvendes med høj grad af sikkerhed. For eksempel kan korrigering af formateringsproblemer eller håndhævelse af grundlæggende datatypebegrænsninger ofte håndteres effektivt af selvhelende dataworkere.

Dog vokser udfordringerne forbundet med automatisk datakorrigering i takt med, at kompleksiteten af datadomænet øges. I domæner med indviklet forretningslogik, komplekse relationer mellem dataentiteter eller domænespecifikke regler og undtagelser, kan selvhelende datateknikker ikke altid fange nuancerne og kan introducere utilsigtede konsekvenser.

Lad os betragte et eksempel på et komplekst domæne: et finansielt handelssystem. I dette domæne involverer dataene forskellige finansielle instrumenter, markedsdata, handelsregler og lovmæssige krav. Relationerne mellem forskellige dataelementer kan være indviklede, og reglerne for datavaliditet og konsistens kan være meget specifikke for domænet.

I et så komplekst domæne ville en selvhelende dataworker, der har til opgave at korrigere uoverensstemmelser i handelsdata, have behov for en dyb forståelse af de domænespecifikke regler og begrænsninger. Den skulle tage hensyn til faktorer som markedsreguleringer, handelsgrænser, risikoberegninger og afviklingsprocedurer. Automatiske korrektioner i denne sammenhæng kan ikke altid fange domænets fulde kompleksitet og kan utilsigtet introducere fejl eller overtræde domænespecifikke regler.

For at håndtere udfordringerne ved domænekompleksitet kan selvhelende datateknikker forbedres ved at inkorporere domænespecifik viden og regler i AI-modellerne og workerne. Dette kan opnås gennem teknikker som:

1. **Domænespecifik Træning**: AI-modellerne, der bruges til selvhelende data, kan dirigeres eller endda finjusteres på domænespecifikke datasæt, der indfanger særegenhederne og reglerne for det specifikke domæne. Ved at eksponere modellerne for repræsentative data og scenarier kan de lære mønstrene, begrænsningerne og undtagelserne, der er specifikke for domænet.
2. **Regelbaserede Begrænsninger**: Selvhelende dataworkere kan udvides med eksplicitte regelbaserede begrænsninger, der koder domænespecifik viden. Disse regler kan defineres af domæneeksperter og integreres i datakorrektionsprocessen. AI-modellerne kan derefter bruge disse regler til at guide deres beslutninger og sikre overholdelse af domænespecifikke krav.
3. **Samarbejde med Domæneeksperter**: I komplekse domæner er det afgørende at involvere domæneeksperter i design og udvikling af selvhelende datateknikker. Domæneeksperter kan bidrage med værdifuld indsigt i dataenes kompleksitet, forretningsreglerne og potentielle særtilfælde. Deres viden kan inkorporeres i AI-modellerne og workerne for at forbedre nøjagtigheden og pålideligheden af automatiske datakorrektioner ved hjælp af Human In The Loop mønstre.
4. **Inkrementel og Iterativ Tilgang**: Når man arbejder med komplekse domæner, er det ofte fordelagtigt at adoptere en inkrementel og iterativ tilgang til selvhelende data. I stedet for at forsøge at automatisere korrektioner for hele domænet på én gang, fokuserer man på specifikke subdomæner eller datakategorier, hvor reglerne og begrænsningerne er velforståede. Gradvist udvider man omfanget af selvhelende teknikker, efterhånden som forståelsen af domænet vokser, og teknikkerne viser sig effektive.

Ved at tage højde for kompleksiteten i datadomænet og inkorporere domænespecifik viden i selvhelende datateknikker kan man skabe balance mellem automatisering og

nøjagtighed. Det er vigtigt at erkende, at selvhelende data ikke er en universel løsning, og at tilgangen bør tilpasses de specifikke krav og udfordringer i hvert domæne.

I komplekse domæner kan en hybrid tilgang, der kombinerer selvhelende datateknikker med menneskelig ekspertise og overvågning, være mest effektiv. Automatiske korrektioner kan håndtere rutineprægede og veldefinerede tilfælde, mens komplekse scenarier eller undtagelser kan markeres til menneskelig gennemgang og indgriben. Denne samarbejdende tilgang sikrer, at fordelene ved automatisering realiseres, samtidig med at den nødvendige kontrol og nøjagtighed i komplekse datadomæner opretholdes.

## Forklarbarhed og Gennemsigtighed

Forklarbarhed henviser til evnen til at forstå og fortolke ræsonnementet bag de beslutninger, der træffes af AI-modeller, mens gennemsigtighed involverer at give klar indsigt i datakorrektionsprocessen.

I mange sammenhænge skal dataændringer være reviderbare og kunne retfærdiggøres. Interessenter, herunder forretningsbrugere, revisorer og tilsynsmyndigheder, kan kræve forklaringer på, hvorfor bestemte datakorrektioner blev foretaget, og hvordan AI-modellerne nåede frem til disse beslutninger. Dette er især afgørende i domæner, hvor datanøjagtighed og -integritet har væsentlige konsekvenser, såsom finans, sundhedsvæsen og juridiske anliggender.

For at imødekomme behovet for forklarbarhed og gennemsigtighed bør selvhelende datateknikker inkorporere mekanismer, der giver indsigt i AI-modellernes beslutningsproces. Dette kan opnås gennem forskellige tilgange:

1. **Tankerække**: Ved at bede modellen om at forklare sin tænkning "højt" før anvendelse af ændringer i data, kan man lettere forstå beslutningsprocessen og kan generere menneskelæsbare forklaringer på de foretagne korrektioner. Kompromisset er en smule mere kompleksitet i adskillelsen af forklaringen fra det strukturerede dataoutput, hvilket kan håndteres ved…

2. **Forklaringsgenerering**: Selvhelende dataarbejdere kan udstyres med evnen til at generere menneskelæsbare forklaringer på de korrektioner, de foretager. Dette kan opnås ved at bede modellen om at outputte sin beslutningsproces som letforståelige forklaringer *integreret i selve dataene*. For eksempel kunne en selvhelende dataarbejder generere en rapport, der fremhæver de specifikke datainkonsistenser den identificerede, de korrektioner den anvendte, og begrundelsen bag disse korrektioner.
3. **Funktionsvægtning**: AI-modeller kan instrueres med information om vigtigheden af forskellige funktioner eller attributter i datakorrektionsprocessen som en del af deres direktiver. Disse direktiver kan derefter eksponeres for menneskelige interessenter. Ved at identificere de nøglefaktorer, der påvirker modellens beslutninger, kan interessenter få indsigt i ræsonnementet bag korrektionerne og vurdere deres gyldighed.
4. **Logning og Revision**: Implementering af omfattende lognings- og revisionsmekanismer er afgørende for at opretholde gennemsigtighed i den selvhelende dataproces. Hver datakorrektion foretaget af AI-modeller bør logges, inklusive de originale data, de korrigerede data og de specifikke handlinger, der er foretaget. Dette revisionsspor muliggør retrospektiv analyse og giver en klar registrering af de ændringer, der er foretaget i dataene.
5. **Menneske-i-kredsløbet-tilgang**: Inkorporering af en menneske-i-kredsløbet-tilgang kan forbedre forklarbarheden og gennemsigtigheden af selvhelende datateknikker. Ved at involvere menneskeeksperter i gennemgang og validering af AI-genererede korrektioner kan organisationer sikre, at korrektionerne er i overensstemmelse med domæneviden og forretningsmæssige krav. Menneskelig overvågning tilføjer et ekstra lag af ansvarlighed og tillader identifikation af potentielle bias eller fejl i AI-modellerne.
6. **Kontinuerlig Overvågning og Evaluering**: Regelmæssig overvågning og evaluering af selvhelende datateknikkers ydeevne er essentielt for at opretholde gennemsigtighed og tillid. Ved at vurdere AI-modellernes nøjagtighed og

effektivitet over tid kan organisationer identificere eventuelle afvigelser eller anomalier og træffe korrigerende foranstaltninger. Kontinuerlig overvågning hjælper med at sikre, at den selvhelende dataproces forbliver pålidelig og tilpasset de ønskede resultater.

Forklarbarhed og gennemsigtighed er kritiske overvejelser ved implementering af selvhelende datateknikker. Ved at give klare forklaringer på datakorrektioner, vedligeholde omfattende revisionsspor og involvere menneskelig overvågning kan organisationer opbygge tillid til den selvhelende dataproces og sikre, at ændringerne i dataene er berettigede og i overensstemmelse med forretningsmålene.

Det er vigtigt at finde en balance mellem fordelene ved automatisering og behovet for gennemsigtighed. Mens selvhelende datateknikker kan forbedre datakvalitet og effektivitet betydeligt, bør det ikke ske på bekostning af at miste overblik og kontrol over datakorrektionsprocessen. Ved at designe selvhelende dataarbejdere med forklarbarhed og gennemsigtighed for øje kan organisationer udnytte AI's kraft samtidig med at opretholde det nødvendige niveau af ansvarlighed og tillid til dataene.

## Utilsigtede Konsekvenser

Mens selvhelende datateknikker sigter mod at forbedre datakvalitet og konsistens, er det afgørende at være opmærksom på potentialet for utilsigtede konsekvenser. Automatiske korrektioner kan, hvis de ikke er omhyggeligt designet og overvåget, utilsigtet ændre betydningen eller konteksten af dataene, hvilket fører til afledte problemer.

En af de primære risici ved selvhelende data er introduktionen af bias eller fejl i datakorrektionsprocessen. AI-modeller kan, ligesom ethvert andet softwaresystem, være underlagt bias, der er til stede i træningsdata eller introduceret gennem algoritmernes design. Hvis disse bias ikke identificeres og afbødes, kan de forplante sig gennem den selvhelende dataproces og resultere i skævvredne eller ukorrekte datamodifikationer.

Tag for eksempel en selvhelbredende dataarbejder, der har til opgave at korrigere uoverensstemmelser i kunders demografiske data. Hvis AI-modellen har lært fordomme

fra historiske data, såsom at forbinde bestemte erhverv eller indkomstniveauer med specifikke køn eller etniciteter, kan den foretage ukorrekte antagelser og ændre dataene på en måde, der forstærker disse fordomme. Dette kan føre til unøjagtige kundeprofiler, fejlagtige forretningsbeslutninger og potentielt diskriminerende resultater.

En anden potentiel utilsigtet konsekvens er tabet af værdifuld information eller kontekst under datakorrigeringsprocessen. Selvhelbredende datateknikker fokuserer ofte på at standardisere og normalisere data for at sikre konsistens. I nogle tilfælde kan de oprindelige data dog indeholde nuancer, undtagelser eller kontekstuel information, som er vigtig for at forstå det fulde billede. Automatiske korrektioner, der blindt gennemtvinger standardisering, kan utilsigtet fjerne eller sløre denne værdifulde information.

Forestil dig for eksempel en selvhelbredende dataarbejder med ansvar for at korrigere uoverensstemmelser i medicinske journaler. Hvis arbejderen støder på en patients sygehistorie med en sjælden tilstand eller en usædvanlig behandlingsplan, kan den forsøge at normalisere dataene til at passe ind i et mere almindeligt mønster. Ved at gøre dette kan den dog miste de specifikke detaljer og den kontekst, der er afgørende for at repræsentere patientens unikke situation præcist. Dette tab af information kan have alvorlige konsekvenser for patientpleje og medicinske beslutninger.

For at mindske risikoen for utilsigtede konsekvenser er det essentielt at tage en proaktiv tilgang ved design og implementering af selvhelbredende datateknikker:

1. **Grundig Test og Validering**: Før selvhelbredende dataarbejdere implementeres i produktion, er det afgørende at teste og validere deres adfærd grundigt mod en række forskellige scenarier. Dette omfatter test med repræsentative datasæt, der dækker forskellige kanttilfælde, undtagelser og potentielle fordomme. Rigoros test hjælper med at identificere og håndtere eventuelle utilsigtede konsekvenser, før de påvirker data i den virkelige verden.
2. **Kontinuerlig Overvågning og Evaluering**: Implementering af kontinuerlige overvågnings- og evalueringsmekanismer er essentielt for at opdage og afbøde

utilsigtede konsekvenser over tid. Regelmæssig gennemgang af resultaterne fra selvhelbredende dataprocesser, analyse af påvirkningen på downstream-systemer og beslutningstagning, samt indsamling af feedback fra interessenter kan hjælpe med at identificere eventuelle negative effekter og igangsætte rettidige korrigerende handlinger. Hvis din organisation har operationelle dashboards, er det sandsynligvis en god idé at tilføje tydeligt synlige metrikker relateret til automatiserede dataændringer. At tilføje alarmer forbundet med store afvigelser fra normal dataændringsaktivitet er sandsynligvis en endnu bedre idé!

3. **Menneskelig Overvågning og Intervention**: Det er afgørende at opretholde menneskelig overvågning og muligheden for at gribe ind i den selvhelbredende dataproces. Mens automatisering kan forbedre effektiviteten markant, er det vigtigt at have menneskelige eksperter til at gennemgå og validere de korrektioner, der foretages af AI-modeller, især inden for kritiske eller følsomme domæner. Menneskelig dømmekraft og domæneekspertise kan hjælpe med at identificere og håndtere eventuelle utilsigtede konsekvenser, der måtte opstå.

4. **Forklarbar AI (XAI) og Gennemsigtighed**: Som diskuteret i det foregående afsnit kan inkorporering af forklarbar AI-teknikker og sikring af gennemsigtighed i den selvhelbredende dataproces hjælpe med at afbøde utilsigtede konsekvenser. Ved at give klare forklaringer på datakorrektioner og vedligeholde omfattende revisionsspor kan organisationer bedre forstå og spore ræsonnementet bag de ændringer, der foretages af AI-modeller.
5. **Inkrementel og Iterativ Tilgang**: Adoption af en inkrementel og iterativ tilgang til selvhelbredende data kan hjælpe med at minimere risikoen for utilsigtede konsekvenser. I stedet for at anvende automatiske korrektioner på hele datasættet på én gang, start med en delmængde af data og udvid gradvist omfanget efterhånden som teknikkerne viser sig effektive og pålidelige. Dette muliggør omhyggelig overvågning og justering undervejs, hvilket reducerer påvirkningen af eventuelle utilsigtede konsekvenser.

6. **Samarbejde og Feedback**: Engagement af interessenter fra forskellige domæner og opmuntring til samarbejde og feedback gennem hele den selvhelbredende dataproces kan hjælpe med at identificere og håndtere utilsigtede konsekvenser. Regelmæssig indhentning af input fra domæneeksperter, dataforbrugere og slutbrugere kan give værdifuld indsigt i den praktiske påvirkning af datakorrektionerne og fremhæve eventuelle oversete problemer.

Ved proaktivt at adressere risikoen for utilsigtede konsekvenser og implementere passende sikkerhedsforanstaltninger kan organisationer udnytte fordelene ved selvhelbredende datateknikker samtidig med at potentielle negative effekter minimeres. Det er vigtigt at tilgå selvhelbredende data som en iterativ og kollaborativ proces, kontinuerligt overvåge, evaluere og forfine teknikkerne for at sikre, at de er i overensstemmelse med de ønskede resultater og opretholder dataenes integritet og pålidelighed.

Når man overvejer brugen af selvhelbredende datamønstre, er det essentielt at evaluere disse faktorer omhyggeligt og afveje fordelene mod de potentielle risici og begrænsninger. I nogle tilfælde kan en hybrid tilgang, der kombinerer automatiske korrektioner med menneskelig overvågning og intervention, være den mest hensigtsmæssige løsning.

Det er også værd at bemærke, at selvhelbredende datateknikker ikke bør ses som en erstatning for robust datavalidering, inputvalidering og fejlhåndteringsmekanismer. Disse grundlæggende praksisser forbliver kritiske for at sikre dataintegritet og sikkerhed. Selvhelbredende data bør ses som en komplementær tilgang, der kan udvide og forbedre disse eksisterende foranstaltninger.

I sidste ende afhænger beslutningen om at anvende selvhelbredende datamønstre af de specifikke krav, begrænsninger og prioriteter i din applikation. Ved omhyggeligt at overveje de ovennævnte betragtninger og tilpasse dem til din applikations mål

og arkitektur kan du træffe velinformerede beslutninger om hvornår og hvordan selvhelbredende datateknikker kan udnyttes effektivt.

# Kontekstuel Indholdsgenering

Mønstre for Kontekstuel Indholdsgenering udnytter kraften i store sprogmodeller (LLMs) til at generere dynamisk og kontekstspecifikt indhold i applikationer. Denne kategori af mønstre anerkender vigtigheden af at levere personaliseret og relevant indhold til brugere baseret på deres specifikke behov, præferencer og endda tidligere og nuværende interaktioner med applikationen.

> I denne tilgangs kontekst refererer "indhold" både til primært indhold (dvs. blogindlæg, artikler osv.) og meta-indhold, såsom anbefalinger til primært indhold.

Mønstre for Kontekstuel Indholdsgenering kan spille en afgørende rolle i at forbedre dine brugeres engagementsniveauer, levere skræddersyede oplevelser og automatisere indholdsopgaver både for dig og dine brugere. Ved at anvende de mønstre, vi beskriver

i dette kapitel, kan du skabe applikationer, der genererer indhold dynamisk og tilpasser sig kontekst og input i realtid.

Mønstrene fungerer ved at integrere LLMs i applikationens output, lige fra brugergrænsefladen (nogle gange omtalt som "chrome"), til e-mails og andre former for notifikationer, såvel som eventuelle indholdspipelines.

Når en bruger interagerer med applikationen eller udløser en specifik indholdsanmodning, opfanger applikationen den relevante kontekst, såsom brugerpræferencer, tidligere interaktioner eller specifikke prompts. Denne kontekstuelle information fødes derefter ind i LLM'en, sammen med eventuelle nødvendige skabeloner eller retningslinjer, og bruges til at producere tekstoutput, som ellers skulle have været enten hardkodet, gemt i en database eller algoritmisk genereret.

Det LLM-genererede indhold kan antage forskellige former, såsom personlige anbefalinger, dynamiske produktbeskrivelser, tilpassede e-mailsvar eller endda hele artikler eller blogindlæg. En af de mest radikale anvendelser af dette indhold, som jeg var pionér for for over et år siden, er dynamisk generering af UI-elementer som formularetiketter, værktøjstips og andre former for forklarende tekst.

## Personalisering

En af de vigtigste fordele ved mønstre for Kontekstuel Indholdsgenering er muligheden for at levere meget personlige oplevelser til brugerne. Ved at generere indhold baseret på brugerspecifik kontekst gør disse mønstre det muligt for applikationer at skræddersy indhold til individuelle brugeres interesser, præferencer og interaktioner.

Personalisering handler om mere end blot at indsætte en brugers navn i generisk indhold. Det involverer udnyttelse af den righoldige kontekst, der er tilgængelig om hver bruger, til at generere indhold, der resonerer med deres specifikke behov og ønsker. Denne kontekst kan omfatte en bred vifte af faktorer, såsom:

1. **Brugerprofiloplysninger**: På det mest generelle niveau for anvendelse af denne teknik kan demografiske data, interesser, præferencer og andre profilattributter bruges til at generere indhold, der er i overensstemmelse med brugerens baggrund og karakteristika.
2. **Adfærdsdata**: En brugers tidligere interaktioner med applikationen, såsom viste sider, klikkede links eller købte produkter, kan give værdifuld indsigt i deres adfærd og interesser. Disse data kan bruges til at generere indholdsforslag, der afspejler deres engagementsmønstre og forudsiger deres fremtidige behov.
3. **Kontekstuelle Faktorer**: Brugerens aktuelle kontekst, såsom deres placering, enhed, tidspunkt på dagen eller endda vejret, kan påvirke indholdsgenereringsprocessen. For eksempel kunne en rejseapplikation have en AI-medarbejder, der er i stand til at generere personlige anbefalinger baseret på brugerens aktuelle placering og de aktuelle vejrforhold.

Ved at udnytte disse kontekstuelle faktorer gør mønstre for Kontekstuel Indholdsgenering det muligt for applikationer at levere indhold, der føles skræddersyet til hver enkelt bruger. Dette niveau af personalisering har flere væsentlige fordele:

1. **Øget Engagement**: Personaliseret indhold fanger brugernes opmærksomhed og holder dem engagerede i applikationen. Når brugerne føler, at indholdet er relevant og taler direkte til deres behov, er de mere tilbøjelige til at bruge mere tid på at interagere med applikationen og udforske dens funktioner.
2. **Forbedret Brugertilfredshed**: Personaliseret indhold viser, at applikationen forstår og tager hensyn til brugerens unikke krav. Ved at levere indhold, der er hjælpsomt, informativt og i overensstemmelse med deres interesser, kan applikationen øge brugertilfredsheden og opbygge en stærkere forbindelse med sine brugere.
3. **Højere Konverteringsrater**: I forbindelse med e-handel eller marketingapplikationer kan personaliseret indhold have betydelig indvirkning på konverteringsrater. Ved at præsentere brugerne for produkter, tilbud

eller anbefalinger, der er skræddersyet til deres præferencer og adfærd, kan applikationen øge sandsynligheden for, at brugerne foretager ønskede handlinger, såsom at foretage et køb eller tilmelde sig en tjeneste.

## Produktivitet

Mønstre for Kontekstuel Indholdsgenering kan markant øge visse former for produktivitet ved at reducere behovet for manuel indholdsgenering og redigering i kreative processer. Ved at udnytte kraften i LLMs kan du generere kvalitetsindhold i stor skala og spare tid og kræfter, som dine indholdsskabere og udviklere ellers skulle have brugt på kedeligt manuelt arbejde.

Traditionelt set skal indholdsproducenter researche, skrive, redigere og formatere indhold for at sikre, at det opfylder applikationens krav og brugerens forventninger. Denne proces kan være tidskrævende og ressourceintensiv, især når mængden af indhold vokser.

Med mønstre for kontekstuel indholdsproduktion kan indholdsproduktionen dog i vid udstrækning automatiseres. LLM'er kan generere sammenhængende, grammatisk korrekt og kontekstuelt relevant indhold baseret på de givne prompter og retningslinjer. Denne automatisering giver flere produktivitetsfordele:

1. **Reduceret manuelt arbejde**: Ved at uddelegere opgaver med indholdsproduktion til LLM'er kan indholdsproducenter fokusere på opgaver på højere niveau såsom indholdsstrategi, idéudvikling og kvalitetssikring. De kan give LLM'en den nødvendige kontekst, skabeloner og retningslinjer og lade den håndtere selve indholdsproduktionen. Dette reducerer den manuelle indsats, der kræves til at skrive og redigere, hvilket gør indholdsproducenter mere produktive og effektive.
2. **Hurtigere indholdsproduktion**: LLM'er kan generere indhold meget hurtigere end menneskelige forfattere. Med de rigtige prompter og retningslinjer kan en

LLM producere flere stykker indhold på få sekunder eller minutter. Denne hastighed gør det muligt for applikationer at generere indhold i et meget hurtigere tempo og dermed følge med brugernes behov og det konstant foranderlige digitale landskab.

> Fører hurtigere indholdsproduktion til en "tragedy of the commons" situation, hvor internettet drukner i indhold, som ingen læser? Desværre tror jeg, at svaret er ja.

3. **Konsistens og kvalitet**: LLM'er kan uden problemer revidere indhold, så det er konsistent i stil, tone og kvalitet. Med klare retningslinjer og eksempler kan visse typer applikationer (f.eks. nyhedsredaktioner, PR osv.) sikre, at deres menneskeskabte indhold stemmer overens med deres brand voice og opfylder de ønskede kvalitetsstandarder. Denne konsistens reducerer behovet for omfattende redigering og revision og sparer tid og kræfter i indholdsproduktionsprocessen.
4. **Iteration og optimering**: Mønstre for kontekstuel indholdsproduktion muliggør hurtig iteration og optimering af indhold. Ved at justere prompterne, skabelonerne eller retningslinjerne, der gives til LLM'en, kan dine applikationer hurtigt generere variationer af indhold og teste forskellige tilgange på en automatiseret måde, som aldrig har været mulig tidligere. Denne iterative proces tillader hurtigere eksperimentering og forfining af indholdsstrategier, hvilket over tid fører til mere effektivt og engagerende indhold. Denne særlige teknik kan være en total game-changer for applikationer som e-handel, der lever og dør baseret på afvisningsrater og engagement

Det er vigtigt at bemærke, at selvom mønstre for kontekstuel indholdsproduktion kan forbedre produktiviteten markant, eliminerer de ikke fuldstændigt behovet for menneskelig involvering. Indholdsproducenter og redaktører spiller stadig en afgørende rolle i at definere den overordnede indholdsstrategi, give vejledning til LLM'en og sikre kvaliteten og hensigtsmæssigheden af det genererede indhold.

Ved at automatisere de mere repetitive og tidskrævende aspekter af indholdsproduktion frigør mønstre for kontekstuel indholdsproduktion værdifuld menneskelig tid og ressourcer, der kan omdirigeres til opgaver med højere værdi. Denne øgede produktivitet gør det muligt for dig at levere mere personaliseret og engagerende indhold til brugerne, samtidig med at arbejdsgangen for indholdsproduktion optimeres.

## Hurtig iteration og eksperimentering

Mønstre for kontekstuel indholdsproduktion gør det muligt hurtigt at iterere og eksperimentere med forskellige indholdsvariationer, hvilket muliggør hurtigere optimering og forfining af din indholdsstrategi. Du kan generere flere versioner af indhold på få sekunder ved blot at justere konteksten, skabelonerne eller retningslinjerne, der gives til modellen.

Denne mulighed for hurtig iteration giver flere centrale fordele:

1. **Test og optimering**: Med muligheden for hurtigt at generere indholdsvariationer kan du nemt teste forskellige tilgange og måle deres effektivitet. For eksempel kan du generere flere versioner af en produktbeskrivelse eller et marketingbudskab, hver tilpasset til et specifikt brugersegment eller en specifik kontekst. Ved at analysere brugerengagementsmetrikker såsom klikrater eller konverteringsrater kan du identificere de mest effektive indholdsvariationer og optimere din indholdsstrategi i overensstemmelse hermed.

2. **A/B-test**: Mønstre for kontekstuel indholdsproduktion muliggør problemfri A/B-test af indhold. Du kan generere to eller flere variationer af indhold og tilfældigt vise dem til forskellige brugergrupper. Ved at sammenligne hver variations ydeevne kan du afgøre, hvilket indhold der resonerer bedst med din målgruppe. Denne datadrevne tilgang gør det muligt for dig at træffe informerede beslutninger og løbende forfine dit indhold for at maksimere brugerengagement og opnå dine ønskede resultater.
3. **Personaliseringseksperimenter**: Hurtig iteration og eksperimentering er særligt værdifuldt, når det kommer til personalisering. Med mønstre for kontekstuel indholdsproduktion kan du hurtigt generere personaliserede indholdsvariationer baseret på forskellige brugersegmenter, præferencer eller adfærd. Ved at eksperimentere med forskellige personaliseringsstrategier kan du identificere de mest effektive tilgange til at engagere individuelle brugere og levere skræddersyede oplevelser.
4. **Tilpasning til Skiftende Tendenser**: Evnen til at iterere og eksperimentere hurtigt gør det muligt at forblive smidig og tilpasse sig skiftende tendenser og brugerpræferencer. Når nye emner, søgeord eller brugeradfærd opstår, kan du hurtigt generere indhold, der er i tråd med disse tendenser. Ved kontinuerligt at eksperimentere og forfine dit indhold kan du forblive relevant og bevare en konkurrencemæssig fordel i det konstant udviklende digitale landskab.
5. **Omkostningseffektiv Eksperimentering**: Traditionel indholdseksperimentering involverer ofte betydelig tid og ressourcer, da indholdsskabere manuelt skal udvikle og teste forskellige variationer. Med mønstre for Kontekstuel Indholdsgenering er omkostningerne ved eksperimentering dog kraftigt reduceret. Store sprogmodeller kan generere indholdsvariationer hurtigt og i stor skala, hvilket giver dig mulighed for at udforske en bred vifte af idéer og tilgange uden at pådrage sig væsentlige omkostninger.

For at få mest muligt ud af hurtig iteration og eksperimentering er det vigtigt at have en veldefineret eksperimenteringsramme på plads. Denne ramme bør omfatte:

- Klare mål og hypoteser for hvert eksperiment
- Passende metrikker og sporingsmekanismer til at måle indholdets ydeevne
- Segmenterings- og målretningsstrategier for at sikre, at relevante indholdsvariationer leveres til de rigtige brugere
- Analyse- og rapporteringsværktøjer til at udlede indsigter fra de eksperimentelle data
- En proces for at inkorporere læring og optimeringer i din indholdsstrategi

Ved at omfavne hurtig iteration og eksperimentering kan du kontinuerligt forfine og optimere dit indhold, så det forbliver engagerende, relevant og effektivt i forhold til at nå din applikations mål. Denne smidige tilgang til indholdsproduktion giver dig mulighed for at være på forkant og levere exceptionelle brugeroplevelser.

## Skalerbarhed og Effektivitet

I takt med at applikationer vokser, og efterspørgslen efter personaliseret indhold stiger, muliggør kontekstuelle indholdsgenereringsmønstre effektiv skalering af indholdsproduktion. Store sprogmodeller kan generere indhold til et stort antal brugere og kontekster samtidigt, uden behov for en proportionel stigning i menneskelige ressourcer. Denne skalerbarhed giver applikationer mulighed for at levere personlige oplevelser til en voksende brugerbase uden at overbelaste deres indholdsproduktionskapacitet.

Bemærk, at kontekstuel indholdsgenering kan bruges effektivt til at internationalisere din applikation "på farten". Faktisk er det præcis det, jeg gjorde ved hjælp af min Instant18n Gem til at levere Olympia på mere end et halvt dusin sprog, selvom vi er mindre end et år gamle.

# AI-drevet Lokalisering

Hvis I tillader mig at prale et øjeblik, mener jeg, at mit Instant18n-bibliotek til Rails-apps er et banebrydende eksempel på "Kontekstuel Indholdsgenering"-mønstret i aktion, der viser det transformative potentiale for AI i applikationsudvikling. Denne gem udnytter kraften fra OpenAIs GPT store sprogmodel til at revolutionere måden, hvorpå internationalisering og lokalisering håndteres i Rails-applikationer.

Traditionelt involverer internationalisering af en Rails-applikation manuel definition af oversættelsesnøgler og tilvejebringelse af tilsvarende oversættelser for hvert understøttet sprog. Denne proces kan være tidskrævende, ressourceintensiv og tilbøjelig til inkonsistenser. Med Instant18n-gemmen er lokaliseringsparadigmet dog fuldstændigt redefineret.

Ved at integrere en stor sprogmodel gør Instant18n-gemmen det muligt at generere oversættelser on-the-fly, baseret på tekstens kontekst og betydning. I stedet for at være afhængig af foruddefinerede oversættelsesnøgler og statiske oversættelser, oversætter gemmen dynamisk tekst ved hjælp af AI's kraft. Denne tilgang tilbyder flere centrale fordele:

1. **Problemfri Lokalisering**: Med Instant18n-gemmen behøver udviklere ikke længere manuelt at definere og vedligeholde oversættelsesfiler for hvert understøttet sprog. Gemmen genererer automatisk oversættelser baseret på den givne tekst og det ønskede målsprog, hvilket gør lokaliseringsprocessen ubesværet og problemfri.
2. **Kontekstuel Nøjagtighed**: AI kan gives tilstrækkelig kontekst til at forstå nuancerne i den tekst, der oversættes. Den kan tage højde for den omgivende kontekst, talemåder og kulturelle referencer for at generere oversættelser, der er præcise, naturligt lydende og kontekstuelt passende.
3. **Omfattende Sprogunderstøttelse**: Instant18n-gemmen udnytter GPT's omfattende viden og sproglige kapaciteter, hvilket muliggør oversættelser

til et omfattende udvalg af sprog. Fra almindelige sprog som spansk og fransk til mere obskure eller fiktive sprog som klingon og elvisk kan gemmen håndtere en bred vifte af oversættelseskrav.

4. **Fleksibilitet og Kreativitet**: Gemmen går ud over traditionelle sprogoversættelser og tillader kreative og utraditionelle lokaliseringsmuligheder. Udviklere kan oversætte tekst til forskellige stilarter, dialekter eller endda fiktive sprog, hvilket åbner nye muligheder for unikke brugeroplevelser og engagerende indhold.
5. **Ydelsesoptimering**: Instant18n-gemmen inkorporerer cache-mekanismer for at forbedre ydelsen og reducere overhead ved gentagne oversættelser. Oversat tekst caches, hvilket gør det muligt at betjene efterfølgende anmodninger om samme oversættelse hurtigt uden behov for redundante API-kald.

Instant18n-gemmen eksemplificerer kraften i "Kontekstuel Indholdsgenering"-mønstret ved at udnytte AI til at generere lokaliseret indhold dynamisk. Den viser, hvordan AI kan integreres i kernefunktionaliteten af en Rails-applikation og transformere den måde, udviklere tilgår internationalisering og lokalisering på.

Ved at eliminere behovet for manuel oversættelseshåndtering og muliggøre oversættelser i realtid baseret på kontekst, sparer Instant18n gem udviklere betydelig tid og kræfter. Det giver dem mulighed for at fokusere på at udvikle kernefunktionaliteten i deres applikation, mens lokaliseringsaspektet håndteres problemfrit og præcist.

## Vigtigheden af Brugertest og Feedback

Til sidst er det vigtigt altid at huske betydningen af brugertest og feedback. Det er afgørende at validere, at kontekstuel indholdsgenering lever op til brugernes forventninger og er i overensstemmelse med applikationens mål. Fortsæt med at iterere og forfine det genererede indhold baseret på brugerindsigter og analyser. Hvis du genererer dynamisk indhold i stor skala, som ville være umuligt at validere manuelt af dig og dit team, bør du overveje at tilføje feedback-mekanismer, der giver

brugerne mulighed for at rapportere indhold, der er mærkeligt eller forkert, sammen med en forklaring af hvorfor. Denne værdifulde feedback kan endda fødes til en AI-medarbejder med opgaven at foretage justeringer i den komponent, der genererede indholdet!

# Generative UI

Opmærksomhed er så eftertragtet i disse dage, at effektivt brugerengagement nu kræver softwareoplevelser, der ikke kun er problemfrie og intuitive, men også i høj grad personligt tilpasset den enkeltes behov, præferencer og kontekst. Som følge heraf står designere og udviklere i stigende grad over for udfordringen med at skabe brugergrænseflader, der kan tilpasse sig og imødekomme hver enkelt brugers unikke behov *i stor skala*.

Generative UI (GenUI) er en virkelig revolutionerende tilgang til design af brugergrænseflader, der udnytter kraften i store sprogmodeller (LLMs) til at skabe højt personaliserede og dynamiske brugeroplevelser i realtid. Jeg ønskede at sikre mig, at jeg i det mindste gav dig en introduktion til GenUI i denne bog, fordi jeg mener, at det er en af de mest lovende nye muligheder, der i øjeblikket eksisterer inden for applikationsdesign og frameworks. Jeg er overbevist om, at dusinvis eller flere nye succesfulde kommercielle og open source-projekter vil dukke op i denne særlige niche.

I sin kerne kombinerer GenUI principperne for Kontekstbaseret Indholdsgenering med avancerede AI-teknikker til dynamisk at generere brugergrænsefladeelementer, såsom tekst, billeder og layouts, baseret på en dyb forståelse af brugerens kontekst, præferencer og mål. GenUI gør det muligt for designere og udviklere at skabe grænseflader, der tilpasser og udvikler sig som reaktion på brugerinteraktioner, hvilket giver et niveau af personalisering, der tidligere var uopnåeligt.

GenUI repræsenterer en fundamental ændring i måden, vi tilgår design af brugergrænseflader. I stedet for at designe til masserne tillader GenUI os at designe til individet. Personaliseret indhold og grænseflader har potentialet til at skabe brugeroplevelser, der resonerer med hver bruger på et dybere niveau, hvilket øger engagement, tilfredshed og loyalitet.

Som en banebrydende teknik er overgangen til GenUI fuld af konceptuelle og praktiske udfordringer. Integration af AI i designprocessen, sikring af at de genererede grænseflader ikke kun er personaliserede, men også brugbare, tilgængelige og aligned med det overordnede brand og brugeroplevelse - alt dette er udfordringer, der gør GenUI til en beskæftigelse for de få, ikke de mange. Derudover rejser involveringen af AI spørgsmål om databeskyttelse, gennemsigtighed og måske endda etiske implikationer.

På trods af udfordringerne har personaliserede oplevelser i stor skala potentialet til at transformere måden, vi interagerer med digitale produkter og tjenester på fuldstændigt. Det åbner muligheder for at skabe inkluderende og tilgængelige grænseflader, der imødekommer brugernes forskellige behov, uanset deres evner, baggrund eller præferencer.

I dette kapitel vil vi udforske konceptet GenUI og undersøge nogle definerende karakteristika, centrale fordele og potentielle udfordringer. Vi begynder med at overveje den mest grundlæggende og tilgængelige form for GenUI: generering af tekstindhold til ellers traditionelt designede og implementerede brugergrænseflader.

# Generering af tekst til brugergrænseflader

Tekstelementer, der findes i din applikations brugergrænseflade-elementer, såsom formularetiketter, værktøjstips og forklarende tekst, er typisk hardcodet ind i skabelonerne eller UI-komponenterne, hvilket giver en konsistent men generisk oplevelse for alle brugere. Ved at bruge mønstre for kontekstbaseret indholdsgenering kan du transformere disse statiske elementer til dynamiske, kontekstbevidste og personaliserede komponenter.

## Personaliserede formularer

Formularer er en allestedsnærværende del af web- og mobilapplikationer og fungerer som det primære middel til at indsamle brugerinput. Traditionelle formularer præsenterer dog ofte en generisk og upersonlig oplevelse med standardetiketter og -felter, der ikke altid stemmer overens med brugerens specifikke kontekst eller behov. Brugere er mere tilbøjelige til at udfylde formularer, der føles skræddersyet til deres behov og præferencer, hvilket fører til højere konverteringsrater og brugertilfredshed.

Det er dog vigtigt at finde en balance mellem personalisering og konsistens. Mens tilpasning af formularer til individuelle brugere kan være gavnlig, er det afgørende at opretholde et niveau af genkendelighed og forudsigelighed. Brugere skal stadig kunne genkende og navigere i formularer let, selv med personaliserede elementer.

Her er nogle personaliserede formular-idéer til inspiration:

### Kontekstuelle feltforslag

GenUI kan analysere brugerens tidligere interaktioner, præferencer og data for at give intelligente feltforslag som forudsigelser. Hvis brugeren for eksempel tidligere har indtastet deres leveringsadresse, kan formularen automatisk udfylde de relevante felter med deres gemte oplysninger. Dette sparer ikke kun tid, men viser også, at applikationen forstår og husker brugerens præferencer.

Vent lige et øjeblik, er denne teknik ikke noget, der kunne gøres uden at involvere AI? Selvfølgelig, men det smukke ved at drive denne type funktionalitet med AI er todelt: 1) hvor nemt det kan være at implementere og 2) hvor modstandsdygtigt det kan være, efterhånden som din brugergrænseflade ændrer og udvikler sig over tid.

Lad os hurtigt lave en service til vores teoretiske ordrehåndteringssystem, som forsøger at udfylde den rigtige leveringsadresse for brugeren på forhånd.

```ruby
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 eksempel er meget forenklet, men burde virke i de fleste tilfælde. Idéen er at lade AI'en gætte på samme måde, som et menneske ville gøre. For at gøre det klart, hvad jeg taler om, lad os se på nogle eksempeldata:

```
Order History:
[
  {"date": "2024-01-03", "description": "garden soil mix",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-01-15", "description": "hardcover fiction novels",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-01-22", "description": "baby diapers", "shipping_address":
   "789 Suburb St, Quietville"},
  {"date": "2024-02-01", "description": "organic vegetables",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-02-17", "description": "mystery thriller book set",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-02-25", "description": "baby wipes",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-03-05", "description": "flower seeds",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-03-20", "description": "biographies",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-03-30", "description": "baby formula",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-04-12", "description": "lawn fertilizer",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-04-22", "description": "science fiction novels",
   "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"}
]
```

Lagde du mærke til mønstret i dataene? Jeg garanterer dig, at dette er let at forstå for en LLM. For at demonstrere dette, lad os spørge GPT-4 hvad den mest sandsynlige leveringsadresse er 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 tænker, at det er at skyde gråspurve med kanoner at bruge en dyr model som GPT-4 til denne opgave, har du ret! Jeg prøvede den samme prompt på Mistral 7B Instruct, og den producerede følgende svar med 75 tokens i sekundet og til den beskedne pris af $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 overhead og omkostninger ved denne teknik det værd for at gøre checkout-oplevelsen mere magisk? For mange online forhandlere, helt bestemt. Og ud fra hvordan det ser ud, vil omkostningerne ved AI-beregning kun falde, især for udbydere af open source model-hosting i et kapløb mod bunden.

Brug en Prompt Template og StructuredIO sammen med Response Fencing for at optimere denne type chat-færdiggørelse.

## Adaptiv feltrækkefølge

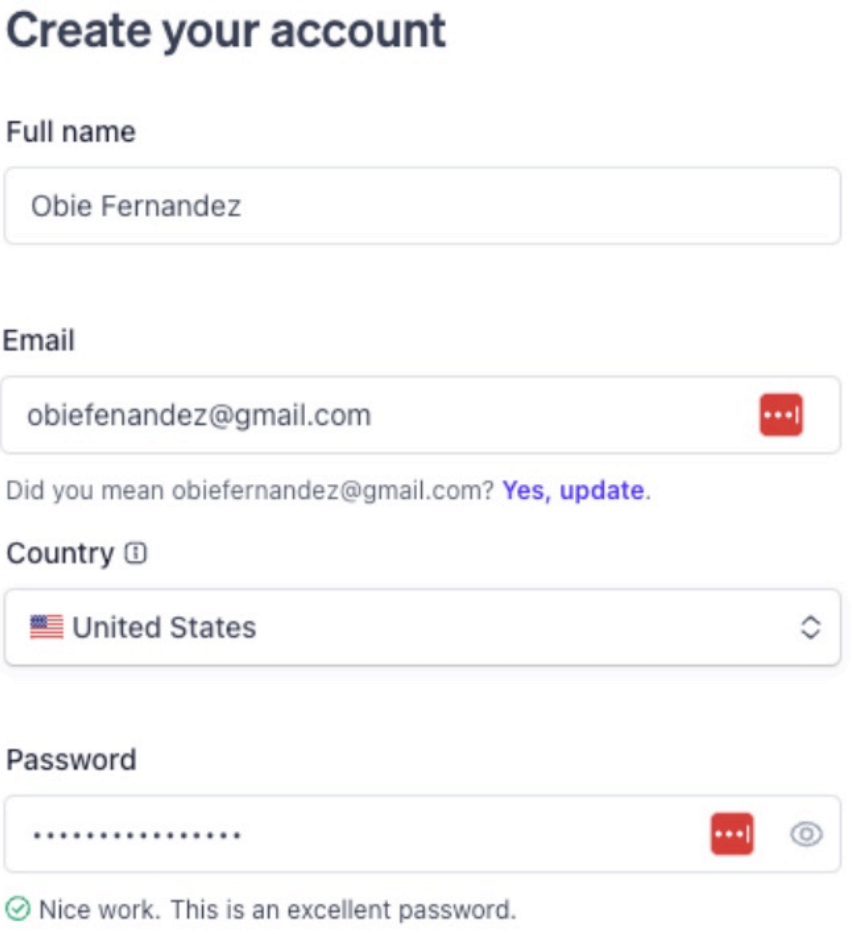
Rækkefølgen, hvori formularfelter præsenteres, kan have betydelig indflydelse på brugerens oplevelse og færdiggørelsesrater. Med GenUI kan du dynamisk justere felternes rækkefølge baseret på brugerens kontekst og vigtigheden af hvert felt. For eksempel, hvis brugeren udfylder en tilmeldingsformular til en fitness-app, kunne formularen prioritere felter relateret til deres træningsmål og præferencer, hvilket gør processen mere relevant og engagerende.

## Personaliseret mikrotekst

Den instruerende tekst, fejlmeddelelser og anden mikrotekst forbundet med formularer kan også personaliseres ved hjælp af GenUI. I stedet for at vise generiske fejlmeddelelser som "Ugyldig e-mailadresse," kan du generere mere hjælpsomme og kontekstuelle beskeder såsom "Indtast venligst en gyldig e-mailadresse for at modtage din ordrebekræftelse." Disse personlige detaljer kan gøre formularoplevelsen mere brugervenlig og mindre frustrerende.

## Personaliseret validering

I forlængelse af Personaliseret mikrotekst, kunne du bruge AI til at validere formularen på måder, der virker magiske. Forestil dig at lade en AI validere en brugerprofilformular, hvor den leder efter potentielle fejl på et *semantisk* niveau.




**Figur 9. Kan du få øje på den semantiske validering?**

## Progressiv afsløring

GenUI kan intelligent afgøre, hvilke formularfelter der er essentielle baseret på brugerens kontekst og gradvist afsløre yderligere felter efter behov. Denne progressive afsløring hjælper med at reducere den kognitive belastning og gør formularudfyldningsprocessen mere håndterbar. For eksempel, hvis en bruger tilmelder

sig et basis-abonnement, kan formularen indledningsvist kun præsentere de essentielle felter, og efterhånden som brugeren skrider frem eller vælger specifikke muligheder, kan yderligere relevante felter introduceres dynamisk.

## Kontekstbevidst forklarende tekst

Værktøjstips bruges ofte til at give yderligere information eller vejledning til brugere, når de holder musen over eller interagerer med specifikke elementer. Med en "Kontekstuel indholdsgenereringstilgang" kan du generere værktøjstips, der tilpasser sig brugerens kontekst og giver relevant information. For eksempel, hvis en bruger udforsker en kompleks funktion, kan værktøjstippet tilbyde personaliserede tips eller eksempler baseret på deres tidligere interaktioner eller færdighedsniveau.

Forklarende tekst, såsom instruktioner, beskrivelser eller hjælpemeddelelser, kan genereres dynamisk baseret på brugerens kontekst. I stedet for at præsentere generiske forklaringer kan du bruge LLMs til at generere tekst, der er skræddersyet til brugerens specifikke behov eller spørgsmål. For eksempel, hvis en bruger har problemer med et bestemt trin i en proces, kan den forklarende tekst give personaliseret vejledning eller fejlfindingstips.

Mikrotekst refererer til de små tekstbidder, der guider brugere gennem din applikation, såsom knapetiketter, fejlmeddelelser eller bekræftelsesprompter. Ved at anvende Kontekstuel indholdsgeneration tilgangen på mikrotekst kan du skabe et adaptivt UI, der reagerer på brugerens handlinger og leverer relevant og hjælpsom tekst. For eksempel, hvis en bruger er ved at udføre en kritisk handling, kan bekræftelsespromten genereres dynamisk for at give en klar og personaliseret besked.

Personaliseret forklarende tekst og værktøjstips kan i høj grad forbedre onboarding-processen for nye brugere. Ved at give kontekstspecifik vejledning og eksempler kan du hjælpe brugere med hurtigt at forstå og navigere i applikationen, reducere indlæringskurven og øge adoptionen.

Dynamiske og kontekstbevidste chrome-elementer kan også få applikationen til at føles mere intuitiv og engagerende. Brugere er mere tilbøjelige til at interagere med og udforske funktioner, når den medfølgende tekst er skræddersyet til deres specifikke behov og interesser.

Indtil nu har vi dækket idéer til at forbedre eksisterende UI-paradigmer med AI, men hvad med at gentænke hvordan brugergrænseflader designes og implementeres på en mere radikal måde?

## Definition af Generativ UI

I modsætning til traditionelt UI-design, hvor designere skaber faste, statiske grænseflader, peger GenUI mod en fremtid, hvor vores software har fleksible, personaliserede oplevelser, der kan udvikle sig og tilpasse sig i realtid. Hver gang vi bruger en AI-drevet samtalegrænseflade, lader vi AI'en tilpasse sig brugerens særlige behov. GenUI tager tingene et skridt videre ved at anvende dette niveau af tilpasningsevne på softwarens *visuelle* grænseflade.

Grunden til at det er muligt at eksperimentere med GenUI-idéer i dag er, at store sprogmodeller allerede forstår programmering, og deres grundlæggende viden omfatter UI-teknologier og frameworks. Spørgsmålet er således, om store sprogmodeller kan bruges til at generere UI-elementer, såsom tekst, billeder, layouts og endda hele grænseflader, der er skræddersyet til hver enkelt bruger. Modellen kunne instrueres i at tage højde for forskellige faktorer, såsom brugerens tidligere interaktioner, udtrykte præferencer, demografiske information og den aktuelle brugskontekst, for at skabe meget personaliserede og relevante grænseflader.

GenUI adskiller sig fra traditionelt brugergrænseflade-design på flere centrale måder:

1. **Dynamisk og Adaptiv**: Traditionelt UI-design involverer skabelsen af faste, statiske grænseflader, der forbliver de samme for alle brugere. I modsætning hertil muliggør GenUI grænseflader, der dynamisk kan tilpasse og ændre sig baseret på brugerbehov og kontekst. Dette betyder, at den samme applikation kan præsentere forskellige grænseflader til forskellige brugere eller endda til den samme bruger i forskellige situationer.
2. **Personalisering i Stor Skala**: Med traditionelt design er det ofte upraktisk at skabe personaliserede oplevelser for hver bruger på grund af den tid og de ressourcer, det kræver. GenUI derimod tillader personalisering i stor skala. Ved at udnytte AI kan designere skabe grænseflader, der automatisk tilpasser sig hver brugers unikke behov og præferencer, uden at skulle manuelt designe og udvikle separate grænseflader for hvert brugersegment.
3. **Fokus på Resultater**: Traditionelt UI-design fokuserer ofte på at skabe visuelt tiltalende og funktionelle grænseflader. Mens disse aspekter stadig er vigtige i GenUI, skifter det primære fokus mod at opnå ønskede brugerresultater. GenUI sigter mod at skabe grænseflader, der er optimeret til hver brugers specifikke mål og opgaver, hvor brugervenlighed og effektivitet prioriteres over rent æstetiske overvejelser.
4. **Kontinuerlig Læring og Forbedring**: GenUI-systemer kan kontinuerligt lære og forbedre sig over tid baseret på brugerinteraktioner og feedback. Når brugere interagerer med de genererede grænseflader, kan AI-modellerne indsamle data om brugeradfærd, præferencer og resultater og bruge denne information til at forfine og optimere fremtidige grænseflade-generationer. Denne iterative læringsproces gør det muligt for GenUI-systemer at blive stadig mere effektive til at opfylde brugernes behov over tid.

Det er vigtigt at bemærke, at GenUI ikke er det samme som AI-assisterede designværktøjer, såsom dem der giver forslag eller automatiserer visse designopgaver. Mens disse værktøjer kan være nyttige til at strømline designprocessen, er de stadig afhængige af designere til at træffe endelige beslutninger og skabe statiske grænseflader.

GenUI involverer derimod, at AI-systemet tager en mere aktiv rolle i den faktiske generering og tilpasning af grænseflader baseret på brugerdata og kontekst.

GenUI repræsenterer et betydeligt skift i hvordan vi tilgår brugergrænseflade-design, hvor vi bevæger os væk fra one-size-fits-all-løsninger og hen imod højt personaliserede, adaptive oplevelser. Ved at udnytte AI's kraft har GenUI potentialet til at revolutionere den måde, vi interagerer med digitale produkter og tjenester på, ved at skabe grænseflader der er mere intuitive, engagerende og effektive for hver enkelt bruger.

# Eksempel

For at illustrere konceptet GenUI, lad os overveje en hypotetisk fitness-applikation kaldet "FitAI". Denne app sigter mod at give personaliserede træningsplaner og ernæringsråd til brugere baseret på deres individuelle mål, fitnessniveauer og præferencer.

I en traditionel UI-design-tilgang ville FitAI måske have et fast sæt skærme og elementer, der er ens for alle brugere. Med GenUI kunne appens grænseflade dog dynamisk tilpasse sig hver brugers unikke behov og kontekst.

Denne tilgang er lidt af en udfordring at forestille sig implementeret i 2024 og har måske ikke engang tilstrækkelig ROI, men det er muligt.

Sådan kunne det fungere:

1. **Onboarding**:

    - I stedet for et standard spørgeskema bruger FitAI en konversations-AI til at indsamle information om brugerens mål, nuværende fitnessniveau og præferencer.
    - Baseret på denne indledende interaktion genererer AI'en et personaliseret dashboard-layout, der fremhæver de funktioner og informationer, der er mest relevante for brugerens mål.

- Nuværende AI-teknologi kunne have et udvalg af skærmkomponenter til rådighed til brug i sammensætningen af det personaliserede dashboard.
- Fremtidig AI-teknologi kunne påtage sig rollen som en erfaren UI-designer og faktisk skabe dashboardet *fra bunden*.

2. **Træningsprogram**:

   - Træningsprogrammets brugergrænseflade tilpasses af AI'en specifikt til brugerens erfaringsniveau og tilgængeligt udstyr.
   - For en nybegynder uden udstyr kan den vise simple kropsvægtsøvelser med detaljerede instruktioner og videoer.
   - For en avanceret bruger med adgang til et fitnesscenter kan den vise mere komplekse rutiner med mindre forklarende indhold.
   - Indholdet af træningsprogrammet er ikke blot filtreret fra en stor samling. Det kan genereres *på stedet* baseret på en vidensbase, der forespørges med kontekst, der omfatter alt kendt om brugeren.

3. **Fremskridtssporing**:

   - Fremskridtssporingens brugergrænseflade udvikler sig baseret på brugerens mål og engagementsmønstre.
   - Hvis en bruger primært fokuserer på vægttab, vil grænsefladen fremhævet vise en vægtudviklingsgraf og statistik over kalorieforbrænding.
   - For en bruger, der opbygger muskler, kan den fremhæve styrkeforøgelser og ændringer i kropskompositionen.
   - AI'en kan tilpasse denne del af applikationen til brugerens faktiske fremskridt. Hvis fremskridtet stopper i en periode, kan appen skifte til en tilstand, hvor den forsøger at få brugeren til at afsløre årsagerne til tilbageslaget for at afhjælpe dem.

4. **Kostvejledning**:

   - Kostdelen tilpasser sig brugerens kostpræferencer og -begrænsninger.

- For en vegansk bruger kan den vise plantebaserede måltidsforslag og proteinkilder.
- For en bruger med glutenintolerance vil den automatisk filtrere glutenholdige fødevarer fra anbefalingerne.
- Igen er indholdet ikke hentet fra en massiv samling af måltidsdata, der gælder for alle brugere, men syntetiseres derimod fra en vidensbase, der indeholder information, der kan tilpasses baseret på brugerens specifikke situation og begrænsninger.
- For eksempel genereres opskrifter med ingrediensspecifikationer, der matcher brugerens konstant skiftende kaloriebehov, efterhånden som deres fitnessniveau og kropsstatistikker udvikler sig.

5. **Motivationselementer**:

   - Appens motiverende indhold og notifikationer er personliggjort baseret på brugerens personlighedstype og respons på forskellige motivationsstrategier.
   - Nogle brugere modtager opmuntrende beskeder, mens andre får mere datadrevet feedback.

I dette eksempel gør GenUI det muligt for FitAI at skabe en højt tilpasset oplevelse for hver bruger, hvilket potentielt øger engagement, tilfredshed og sandsynligheden for at nå træningsmål. Grænsefladeelementer, indhold og endda appens "personlighed" tilpasser sig for bedst at tjene hver enkelt brugers behov og præferencer.

## Skiftet til resultatsorienteret design

GenUI repræsenterer et fundamentalt skift i tilgangen til brugergrænsefladedesign, der bevæger sig fra et fokus på at skabe specifikke grænsefladeelementer til en mere holistisk, resultatorienteret tilgang. Dette skift har flere vigtige implikationer:

1. **Fokus på brugermål**:

- Designere vil skulle tænke dybere over brugermål og ønskede resultater frem for specifikke grænsefladekomponenter.
- Vægten vil være på at skabe systemer, der kan generere grænseflader, som hjælper brugere med at nå deres mål effektivt.
- Nye UI-frameworks vil opstå, der giver AI-baserede designere de værktøjer, de har brug for til at kunne generere brugeroplevelser *på stedet* og *fra bunden* i stedet for baseret på foruddefinerede skærmspecifikationer.

2. **Designeres ændrede rolle**:

   - Designere vil overgå fra at skabe faste layouts til at definere regler, begrænsninger og retningslinjer, som AI-systemer skal følge, når de genererer grænseflader.
   - De vil skulle udvikle færdigheder inden for områder som dataanalyse, AI prompt-udvikling og systemtænkning for effektivt at guide GenUI-systemer.

3. **Vigtigheden af brugerundersøgelser**:

   - Brugerundersøgelser bliver endnu mere kritiske i en GenUI-kontekst, da designere skal forstå ikke kun brugerpræferencer, men også hvordan disse præferencer og behov ændrer sig i forskellige sammenhænge.
   - Kontinuerlig brugertest og feedback-loops vil være essentielle for at forfine og forbedre AI'ens evne til at generere effektive grænseflader.

4. **Design for variabilitet**:

   - I stedet for at skabe en enkelt "perfekt" grænseflade vil designere skulle overveje flere mulige variationer og sikre, at systemet kan generere passende grænseflader til forskellige brugerbehov.
   - Dette omfatter design til grænsetilfælde og sikring af, at de genererede grænseflader opretholder brugervenlighed og tilgængelighed på tværs af forskellige konfigurationer.

- Produktdifferentiering får nye dimensioner, der involverer divergerende perspektiver på brugerpsykologi og udnyttelse af unikke datasæt og videnbaser, der ikke er tilgængelige for konkurrenter.

## Udfordringer og overvejelser

Mens GenUI tilbyder spændende muligheder, præsenterer det også flere udfordringer og overvejelser:

1. **Tekniske begrænsninger**:
    - Nuværende AI-teknologi har, selvom den er avanceret, stadig begrænsninger i forhold til at forstå komplekse brugerintentioner og generere ægte kontekstbevidste grænseflader.
    - Ydelsesproblemer relateret til realtidsgenerering af grænsefladeelementer, især på mindre kraftfulde enheder.
2. **Datakrav**:
    - Afhængigt af anvendelsesformålet kan effektive GenUI-systemer kræve betydelige mængder brugerdata for at generere personaliserede brugergrænseflader.
    - Udfordringerne ved etisk indsamling af autentiske brugerdata rejser bekymringer om databeskyttelse og sikkerhed, samt potentielle skævheder i de data, der bruges til at træne GenUI-modeller.
3. **Brugervenlighed og Konsistens**:
    - I hvert fald indtil praksissen bliver udbredt, kan en applikation med konstant skiftende brugergrænseflader føre til brugervenligheds-problemer, da brugere kan have svært ved at finde velkendte elementer eller navigere effektivt.

- Det vil være afgørende at finde en balance mellem personalisering og opretholdelse af en konsistent, lærbar brugergrænseflade.

4. **Overafhængighed af AI**:

   - Der er en risiko for overdelegering af designbeslutninger til AI-systemer, hvilket potentielt kan føre til uinspirerede, problematiske eller simpelthen defekte grænseflade-valg.
   - Menneskelig overvågning og muligheden for at tilsidesætte AI-genererede designs vil fortsat være vigtig i den overskuelige fremtid.

5. **Tilgængelighedsbekymringer**:

   - At sikre at dynamisk genererede brugergrænseflader forbliver tilgængelige for brugere med handicap præsenterer helt nye udfordringer, hvilket er bekymrende i betragtning af det dårlige niveau af tilgængelighedsoverholdelse, som typiske systemer udviser.
   - På den anden side kan AI-designere implementeres med *indbygget* fokus på tilgængelighed og muligheder for at bygge tilgængelige grænseflader på farten, ligesom de bygger brugergrænseflader til ikke-handicappede brugere.
   - Under alle omstændigheder bør GenUI-systemer designes med robuste tilgængelighedsretningslinjer og testprocesser.

6. **Brugertillid og Gennemsigtighed**:

   - Brugere kan føle sig utilpasse med brugergrænseflader, der synes at "vide for meget" om dem eller ændrer sig på måder, de ikke forstår.
   - At skabe gennemsigtighed omkring hvordan og hvorfor brugergrænseflader personaliseres vil være vigtigt for at opbygge brugertillid.

# Fremtidsudsigter og Muligheder

Fremtiden for Generativ UI (GenUI) rummer et enormt potentiale for at revolutionere måden, hvorpå vi interagerer med digitale produkter og tjenester. I takt med at denne teknologi fortsætter med at udvikle sig, kan vi forvente et skelsættende skift i hvordan brugergrænseflader designes, implementeres og opleves. Jeg tror, at GenUI er det fænomen, der endelig vil skubbe vores software ind i det område, der nu betragtes som science fiction.

Et af de mest spændende aspekter ved GenUI er dets potentiale til at forbedre tilgængelighed i en skala, der går ud over blot at sikre, at personer med alvorlige handicap ikke er fuldstændig udelukket fra at bruge din software. Ved automatisk at tilpasse brugergrænseflader til individuelle brugerbehov kunne GenUI gøre digitale oplevelser mere inkluderende end nogensinde før. Forestil dig brugergrænseflader, der problemfrit justerer sig for at give større tekst til yngre eller synshæmmede brugere eller forenklede layouts til personer med kognitive udfordringer, alt sammen uden at kræve manuel konfiguration eller separate "tilgængelige" versioner af applikationer.

Personaliseringsmulighederne i GenUI vil sandsynligvis drive øget brugerengagement, tilfredshed og loyalitet på tværs af en bred vifte af digitale produkter. Efterhånden som brugergrænseflader bliver mere afstemte med individuelle præferencer og adfærd, vil brugere finde digitale oplevelser mere intuitive og behagelige, hvilket potentielt kan føre til dybere og mere meningsfulde interaktioner med teknologi.

GenUI har også potentialet til at transformere onboarding-processen for nye brugere. Ved at skabe intuitive, personaliserede førstegangsoplevelser, der hurtigt tilpasser sig hver brugers ekspertiseniveau, kunne GenUI markant reducere indlæringskurven forbundet med nye applikationer. Dette kunne føre til hurtigere adoptionsrater og øget brugerselvsikkerhed i udforskningen af nye funktioner og funktionaliteter.

En anden spændende mulighed er GenUI's evne til at opretholde en konsistent brugeroplevelse på tværs af forskellige enheder og platforme, mens der optimeres for

hver specifik brugskontekst. Dette kunne løse den langvarige udfordring med at levere sammenhængende oplevelser på tværs af et stadig mere fragmenteret enhedslandskab, fra smartphones og tablets til stationære computere og fremspirende teknologier som AR-briller.

Den datadrevne natur af GenUI åbner muligheder for hurtig iteration og forbedring i UI-design. Ved at indsamle realtidsdata om, hvordan brugere interagerer med genererede brugergrænseflader, kan designere og udviklere få hidtil usete indsigter i brugeradfærd og præferencer. Denne feedback-loop kunne føre til kontinuerlige forbedringer i UI-design, drevet af faktiske brugsmønstre frem for antagelser eller begrænset brugertest.

For at forberede sig på dette skift vil designere skulle udvikle deres færdigheder og tankesæt. Fokus vil skifte fra at skabe faste layouts til at udvikle omfattende designsystemer og retningslinjer, der kan informere AI-drevet grænsefladegenerering. Designere vil skulle opbygge en dyb forståelse af dataanalyse, AI-teknologier og systemtænkning for effektivt at kunne guide GenUI-systemer.

Desuden, efterhånden som GenUI udvisker grænserne mellem design og teknologi, vil designere skulle samarbejde tættere med udviklere og data scientists. Denne tværfaglige tilgang vil være afgørende i skabelsen af GenUI-systemer, der ikke kun er visuelt tiltalende og brugervenlige, men også teknisk robuste og etisk forsvarlige.

De etiske konsekvenser af GenUI vil også komme i forgrunden, efterhånden som teknologien modnes. Designere vil spille en afgørende rolle i udviklingen af rammer for ansvarlig brug af kunstig intelligens i grænseflade-design, hvor de sikrer, at personalisering forbedrer brugeroplevelser uden at kompromittere privatlivets fred eller manipulere brugeradfærd på uetiske måder.

Når vi ser mod fremtiden, præsenterer GenUI både spændende muligheder og betydelige udfordringer. Det har potentialet til at skabe mere intuitive, effektive og tilfredsstillende digitale oplevelser for brugere over hele verden. Selvom det vil kræve, at designere tilpasser sig og tilegner sig nye færdigheder, giver det også en hidtil uset mulighed for at forme fremtiden for menneske-computer-interaktion på dybdegående og meningsfulde

måder. Rejsen mod fuldt realiserede GenUI-systemer vil uden tvivl være kompleks, men de potentielle gevinster i form af forbedrede brugeroplevelser og digital tilgængelighed gør det til en fremtid, der er værd at stræbe efter.

# Intelligent arbejdsgangsorkestrerering

Inden for applikationsudvikling spiller arbejdsgange en afgørende rolle i at definere, hvordan opgaver, processer og brugerinteraktioner struktureres og udføres. I takt med at applikationer bliver mere komplekse, og brugerforventningerne fortsætter med at stige, bliver behovet for intelligent og adaptiv arbejdsgangsorkestrerering stadig mere åbenlyst.

Tilgangen med "Intelligent arbejdsgangsorkestrerering" fokuserer på at udnytte AI-komponenter til dynamisk at orkestrere og optimere komplekse arbejdsgange i applikationer. Målet er at skabe applikationer, der er mere effektive, responsive og tilpasningsdygtige i forhold til realtidsdata og kontekst.

I dette kapitel vil vi udforske de centrale principper og mønstre, der understøtter den intelligente arbejdsgangsorkestreringstilgang. Vi vil undersøge, hvordan AI kan bruges

til intelligent at dirigere opgaver, automatisere beslutningstagning og dynamisk tilpasse arbejdsgange baseret på forskellige faktorer såsom brugeradfærd, systemydeevne og forretningsregler. Gennem praktiske eksempler og virkelige scenarier vil vi demonstrere AI's transformative potentiale i at strømline og optimere applikationers arbejdsgange.

Uanset om du bygger virksomhedsapplikationer med komplekse forretningsprocesser eller forbrugerrettede applikationer med dynamiske brugerrejser, vil mønstrene og teknikkerne, der diskuteres i dette kapitel, udruste dig med den viden og de værktøjer, der skal til for at skabe intelligente og effektive arbejdsgange, som forbedrer den overordnede brugeroplevelse og skaber forretningsværdi.

## Forretningsmæssigt behov

Traditionelle tilgange til arbejdsgangsstyring er ofte afhængige af foruddefinerede regler og statiske beslutningstræer, som kan være rigide, ufleksible og ude af stand til at håndtere moderne applikationers dynamiske natur.

Overvej et scenarie, hvor en e-handelsapplikation skal håndtere en kompleks ordreekspeditionsproces. Arbejdsgangen kan involvere flere trin såsom ordrevalidering, lagerkontrol, betalingsbehandling, forsendelse og kundenotifikationer. Hvert trin kan have sine egne regler, afhængigheder, eksterne integrationer og undtagelseshåndteringsmekanismer. At administrere sådan en arbejdsgang manuelt eller gennem hardkodet logik kan hurtigt blive besværligt, fejlbehæftet og svært at vedligeholde.

Desuden kan arbejdsgangen, efterhånden som applikationen skalerer og antallet af samtidige brugere vokser, have behov for at tilpasse og optimere sig selv baseret på realtidsdata og systemydeevne. For eksempel kan applikationen under perioder med spidsbelastning have behov for dynamisk at justere arbejdsgangen for at prioritere bestemte opgaver, allokere ressourcer effektivt og sikre en gnidningsfri brugeroplevelse.

Det er her tilgangen med "Intelligent arbejdsgangsorkestrerering" kommer ind i billedet.

Ved at udnytte AI-komponenter kan udviklere skabe arbejdsgange, der er intelligente, adaptive og selvoptimerende. AI kan analysere store mængder data, lære af tidligere erfaringer og træffe informerede beslutninger i realtid for at orkestrere arbejdsgangen effektivt.

## Centrale fordele

1. **Øget effektivitet**: AI kan optimere opgaveallokering, ressourceudnyttelse og arbejdsgangseksekvering, hvilket fører til hurtigere behandlingstider og forbedret samlet effektivitet.
2. **Tilpasningsevne**: AI-drevne arbejdsgange kan dynamisk tilpasse sig skiftende forhold, såsom udsving i brugerefterspørgsel, systemydeevne eller forretningskrav, hvilket sikrer, at applikationen forbliver responsiv og robust.
3. **Automatiseret beslutningstagning**: AI kan automatisere komplekse beslutningsprocesser inden for arbejdsgangen, reducere manuel indgriben og minimere risikoen for menneskelige fejl.
4. **Personalisering**: AI kan analysere brugeradfærd, præferencer og kontekst for at personalisere arbejdsgangen og levere skræddersyede oplevelser til individuelle brugere.
5. **Skalerbarhed**: AI-drevne arbejdsgange kan skalere problemfrit for at håndtere stigende mængder af data og brugerinteraktioner uden at gå på kompromis med ydeevne eller pålidelighed.

I de følgende afsnit vil vi udforske de centrale mønstre og teknikker, der muliggør implementeringen af intelligente arbejdsgange og vise virkelige eksempler på, hvordan AI transformerer arbejdsgangsstyring i moderne applikationer.

# Centrale mønstre

For at implementere intelligent arbejdsgangsorkestrerering i applikationer kan udviklere udnytte flere centrale mønstre, der udnytter AI's kraft. Disse mønstre giver en struktureret tilgang til at designe og administrere arbejdsgange, hvilket gør det muligt for applikationer at tilpasse, optimere og automatisere processer baseret på realtidsdata og kontekst. Lad os udforske nogle af de grundlæggende mønstre i intelligent arbejdsgangsorkestrerering.

## Dynamisk opgavefordeling

Dette mønster involverer brugen af AI til intelligent at dirigere opgaver inden for en arbejdsgang baseret på forskellige faktorer såsom opgaveprioritet, ressourcetilgængelighed og systemydeevne. AI-algoritmer kan analysere karakteristikaene for hver opgave, overveje systemets aktuelle tilstand og træffe informerede beslutninger om at tildele opgaver til de mest passende ressourcer eller behandlingsveje. Dynamisk opgavefordeling sikrer, at opgaver distribueres og udføres effektivt, hvilket optimerer den samlede arbejdsgangsydeevne.

```
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, # ...
```

```
16    function :assign_task_to_resource, # ...
17
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
```

Bemærk løkken, der er skabt af `while`-udtrykket på linje 29, som fortsætter med at spørge AI'en, indtil opgaven er tildelt. På linje 35 gemmer vi transkriptionen af opgaven til senere analyse og fejlfinding, hvis det bliver nødvendigt.

## Kontekstbaseret Beslutningstagning

Du kan bruge meget lignende kode til at træffe kontekstbevidste beslutninger i en arbejdsgang. Ved at analysere relevante datapunkter såsom brugerindstillinger, historiske mønstre og realtidsinput kan AI-komponenter bestemme den mest hensigtsmæssige handlingsvej ved hvert beslutningspunkt i arbejdsgangen. Tilpas din arbejdsgangs adfærd baseret på den specifikke kontekst for hver bruger eller scenario, og levér personlige og optimerede oplevelser.

### Adaptiv Arbejdsgangssammensætning

Dette mønster fokuserer på dynamisk at sammensætte og justere arbejdsgange baseret på skiftende krav eller forhold. AI kan analysere arbejdsgangens nuværende tilstand, identificere flaskehalse eller ineffektivitet og automatisk modificere arbejdsgangens struktur for at optimere ydeevnen. Adaptiv arbejdsgangssammensætning tillader applikationer at udvikle sig kontinuerligt og forbedre deres processer uden at kræve manuel indgriben.

## Håndtering og Genopretning af Undtagelser

Håndtering og genopretning af undtagelser er kritiske aspekter af intelligent arbejdsgangsorkesterering. Når man arbejder med AI-komponenter og komplekse arbejdsgange, er det essentielt at forudse og håndtere undtagelser elegant for at sikre systemets stabilitet og pålidelighed.

Her er nogle vigtige overvejelser og teknikker til håndtering og genopretning af undtagelser i intelligente arbejdsgange:

1. **Undtagelsespropagering**: Implementér en konsistent tilgang til at propagere undtagelser på tværs af arbejdsgangskomponenter. Når en undtagelse opstår inden for en komponent, bør den fanges, logges og propageres til orkestratoren eller en diskret komponent ansvarlig for at håndtere undtagelser. Idéen er at centralisere undtagelseshåndtering og forhindre, at undtagelser bliver stiltiende opslugt, samt åbne muligheder for Intelligent Fejlhåndtering.
2. **Gentagelsesmekanismer**: Gentagelsesmekanismer hjælper med at forbedre arbejdsgangens robusthed og håndtere midlertidige fejl elegant. Det er absolut en god idé at implementere gentagelsesmekanismer for forbigående eller genoprettelige undtagelser, såsom problemer med netværksforbindelse eller utilgængelighed af ressourcer, som automatisk kan forsøges igen

efter en specificeret forsinkelse. At have en AI-drevet orkestrator eller undtagelseshåndtering betyder, at dine gentagelsesstrategier ikke behøver at være mekaniske i deres natur eller afhængige af faste algoritmer som eksponentiel tilbagefald. Du kan overlade håndteringen af gentagelsen til AI-komponentens "skøn", som er ansvarlig for at beslutte, hvordan undtagelsen skal håndteres.

3. **Tilbagefaldsstrategier**: Hvis en AI-komponent ikke kan levere et gyldigt svar eller støder på en fejl—hvilket er en almindelig hændelse givet dens banebrydende natur—skal der være en tilbagefaldsmekanisme på plads for at sikre, at arbejdsgangen kan fortsætte. Dette kan involvere brug af standardværdier, alternative algoritmer eller en Menneske I Loopet til at træffe beslutninger og holde arbejdsgangen i gang.
4. **Kompenserende Handlinger**: Orkestratorens direktiver bør inkludere instruktioner om kompenserende handlinger til at håndtere undtagelser, der ikke kan løses automatisk. Kompenserende handlinger er trin, der tages for at fortryde eller afbøde virkningerne af en mislykket operation. For eksempel, hvis et betalingsprocesseringstrin mislykkes, kunne en kompenserende handling være at tilbagerulle transaktionen og underrette brugeren. Kompenserende handlinger hjælper med at opretholde datakonsistens og integritet i tilfælde af undtagelser.
5. **Undtagelsesovervågning og -alarmering**: Opsæt overvågnings- og alarmeringsmekanismer til at opdage og underrette relevante interessenter om kritiske undtagelser. Orkestratoren kan gøres opmærksom på tærskler og regler for at udløse alarmer, når undtagelser overskrider visse grænser, eller når specifikke typer af undtagelser opstår. Dette muliggør proaktiv identifikation og løsning af problemer, før de påvirker det samlede system.

Her er et eksempel på håndtering og genopretning af undtagelser i en Ruby-arbejdsgangskomponent:

```
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 eksempel kontrollerer `InventoryManager`-komponenten tilgængeligheden af et produkt for en given ordre. Hvis den tilgængelige mængde er utilstrækkelig, udløser den en `InsufficientInventoryError`. Undtagelsen bliver fanget, logget, og en gentagelsesmekanisme implementeres. Hvis grænsen for gentagelsesforsøg overskrides, falder komponenten tilbage til manuel indgriben ved at underrette en administrator.

Ved at implementere robust undtagelseshåndtering og genopretningsmekanismer, kan du sikre, at dine intelligente arbejdsgange er robuste, vedligeholdelsesvenlige og i stand

til at håndtere uventede situationer på en elegant måde.

Disse mønstre danner grundlaget for intelligent arbejdsgangsorkestering og kan kombineres og tilpasses til at opfylde de specifikke krav i forskellige applikationer. Ved at udnytte disse mønstre kan udviklere skabe arbejdsgange, der er fleksible, robuste og optimerede med hensyn til ydeevne og brugeroplevelse.

I det næste afsnit vil vi undersøge, hvordan disse mønstre kan implementeres i praksis ved hjælp af eksempler fra den virkelige verden og kodestykker for at illustrere integrationen af AI-komponenter i workflowstyring.

# Implementering af Intelligent Arbejdsgangsorkestering i Praksis

Nu hvor vi har udforsket de vigtigste mønstre i intelligent arbejdsgangsorkestering, lad os dykke ned i, hvordan disse mønstre kan implementeres i applikationer fra den virkelige verden. Vi vil give praktiske eksempler og kodestykker for at illustrere integrationen af AI-komponenter i workflowstyring.

## Intelligent Ordrebehandler

Lad os dykke ned i et praktisk eksempel på implementering af intelligent arbejdsgangsorkestering ved hjælp af en AI-drevet `OrderProcessor`-komponent i en Ruby on Rails e-handelsapplikation. `OrderProcessor`en realiserer Process Manager Enterprise Integration-konceptet, som vi først mødte i Kapitel 3, da vi diskuterede Multitude of Workers. Komponenten vil være ansvarlig for at administrere ordreekspeditionsarbejdsgangen, træffe rutningsbeslutninger baseret på mellemliggende resultater og orkesterere udførelsen af forskellige behandlingstrin.

Ordreekspeditionsprocessen involverer flere trin såsom ordrevalidering, lagerkontrol, betalingsbehandling og forsendelse. Hvert trin er implementeret som en separat arbejderproces, der udfører en specifik opgave og returnerer resultatet til `OrderProcessoren`. Trinnene er ikke obligatoriske og behøver ikke engang nødvendigvis at blive udført i en præcis rækkefølge.

Her er et eksempel på implementering af `OrderProcessoren`. Den har to mixins fra Raix. Den første (`ChatCompletion`) giver den mulighed for at udføre chat completion, hvilket er det, der gør dette til en AI-komponent. Den anden (`FunctionDispatch`) muliggør function calling fra AI'en, hvilket tillader den at reagere på en prompt med en funktionsinvokering i stedet for en tekstbesked.

Arbejderfunktionerne (`validate_order`, `check_inventory`, et al) delegerer til deres respektive arbejderklasser, som kan være AI- eller ikke-AI-komponenter, med det eneste krav værende, at de returnerer resultaterne af deres arbejde i et format, der kan repræsenteres som en streng.

Som med alle andre eksempler i denne del af bogen er denne kode praktisk talt pseudo-kode og er kun meant til at formidle mønsterets betydning og inspirere dine egne kreationer. Komplette beskrivelser af mønstre og fuldstændige kodeeksempler er inkluderet 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 eksemplet initialiseres OrderProcessor med et ordreobjekt og vedligeholder en transskription af arbejdsgangens udførelse i det typiske samtaleformat, som er karakteristisk for store sprogmodeller. AI'en får fuld kontrol over orchestreringen af de forskellige behandlingstrin, såsom ordrevalidering, lagerkontrol, betalingsbehandling og forsendelse.

Hver gang `chat_completion`-metoden kaldes, sendes transskriptionen til AI'en, så den kan levere en færdiggørelse som et funktionskald. Det er helt op til AI'en at analysere resultatet af det foregående trin og bestemme den passende handling. For eksempel, hvis lagerkontrollen afslører lave lagerniveauer, kan `OrderProcessor` planlægge en genopfyldningsopgave. Hvis betalingsbehandlingen mislykkes, kan den igangsætte et nyt forsøg eller underrette kundesupport.

> Eksemplet ovenfor har ikke definerede funktioner til genopfyldning eller underretning af kundesupport, men det kunne det sagtens have.

Transskriptionen vokser hver gang en funktion kaldes og fungerer som en registrering af arbejdsgangens udførelse, herunder resultaterne af hvert trin og AI-genererede instruktioner til de næste trin. Denne transskription kan bruges til fejlfinding, revision og til at give indblik i ordreafviklingsprocessen.

Ved at udnytte AI i `OrderProcessor` kan e-handelsapplikationen dynamisk tilpasse arbejdsgangen baseret på realtidsdata og håndtere undtagelser intelligent. AI-komponenten kan træffe velinformerede beslutninger, optimere arbejdsgangen og sikre en problemfri ordrebehandling selv i komplekse scenarier.

Det faktum, at det eneste krav til arbejdsprocesserne er at returnere et forståeligt output, som AI'en kan overveje, når den beslutter, hvad der skal gøres næste gang, kan få dig til at indse, hvordan denne tilgang kan reducere det input/output-kortlægningsarbejde, der typisk er involveret, når forskellige systemer skal integreres med hinanden.

## Intelligent Indholdsmoderator

Sociale medieapplikationer kræver generelt mindst minimal indholdsmoderation for at sikre et sikkert og sundt fællesskab. Dette eksempel på en `ContentModerator`-komponent udnytter AI til intelligent at orchestrere moderationsarbejdsgangen ved at træffe beslutninger baseret på indholdets karakteristika og resultaterne af forskellige moderationstrin.

Moderationsprocessen involverer flere trin såsom tekstanalyse, billedgenkendelse, vurdering af brugerens omdømme og manuel gennemgang. Hvert trin er implementeret som en separat arbejdsproces, der udfører en specifik opgave og returnerer resultatet til `ContentModerator`.

Her er et eksempel på implementeringen af `ContentModerator`:

```
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
```

```
83
84    def reject_content
85      @content.update!(status: 'rejected', transcript: @transcript)
86    end
87
88    private
89
90    def continue_with(result)
91      @transcript << { function: result }
92      complete(@transcript)
93    end
94  end
```

I dette eksempel initialiseres `ContentModerator` med et indholdsobjekt og vedligeholder en moderationslog i samtaleformat. AI-komponenten har fuld kontrol over moderationsarbejdsgangen og beslutter, hvilke trin der skal udføres baseret på indholdets karakteristika og resultaterne af hvert trin.

De tilgængelige arbejderfunktioner, som AI'en kan kalde, omfatter `analyze_text`, `recognize_image`, `assess_user_reputation` og `escalate_to_manual_review`. Hver funktion delegerer opgaven til en tilsvarende arbejderproces (`TextAnalysisWorker`, `ImageRecognitionWorker` osv.) og tilføjer resultatet til moderationsloggen, med undtagelse af eskaleringsfunktionen, som fungerer som en sluttilstand. Endelig fungerer funktionerne `approve_content` og `reject_content` også som sluttilstande.

AI-komponenten analyserer indholdet og fastlægger den passende handling. Hvis indholdet indeholder billedhenvisninger, kan den kalde `recognize_image`-arbejderen for at få hjælp til en visuel gennemgang. Hvis nogen arbejder advarer om potentielt skadeligt indhold, kan AI'en beslutte at eskalere indholdet til manuel gennemgang eller blot afvise det med det samme. Men afhængigt af advarslens alvorlighed kan AI'en vælge at bruge resultaterne af brugerens omdømmevurdering til at beslutte, hvordan den skal håndtere indhold, som den ellers er usikker på. Afhængigt af anvendelsesscenariet har betroede brugere måske mere spillerum i forhold til, hvad de kan dele. Og så videre, og så videre...

Som med det tidligere eksempel med processtyring fungerer moderationsloggen som en registrering af arbejdsgangens udførelse, herunder resultaterne af hvert trin og de AI-genererede beslutninger. Denne log kan bruges til revision, gennemsigtighed og forbedring af moderationsprocessen over tid.

Ved at udnytte AI i `ContentModerator` kan sociale medie-applikationen dynamisk tilpasse moderationsarbejdsgangen baseret på indholdets karakteristika og intelligent håndtere komplekse moderationsscenarier. AI-komponenten kan træffe velinformerede beslutninger, optimere arbejdsgangen og sikre en sikker og sund fællesskabsoplevelse.

Lad os udforske to eksempler mere, der demonstrerer prædiktiv opgaveplanlægning og fejlhåndtering og -genopretning inden for rammerne af intelligent arbejdsgangsorkestrating.

## Prædiktiv opgaveplanlægning i et kundesupportsystem

I en kundesupportapplikation bygget med Ruby on Rails er effektiv håndtering og prioritering af supporthenvendelser afgørende for at yde rettidig assistance til kunder. `SupportTicketScheduler`-komponenten udnytter AI til prædiktivt at planlægge og tildele supporthenvendelser til tilgængelige supportmedarbejdere baseret på forskellige faktorer såsom henvendelsens hastende karakter, medarbejderens ekspertise og arbejdsbyrde.

```ruby
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 }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  def functions
    [
      {
        name: "analyze_ticket_urgency",
        # ...
      },
      {
        name: "list_available_agents",
        description: "Includes expertise of available agents",
        # ...
      },
      {
        name: "predict_agent_workload",
        description: "Uses historical data to predict upcoming workloads",
        # ...
      },
      {
        name: "assign_ticket_to_agent",
        # ...
```

```
      },
      {
        name: "reschedule_ticket",
        # ...
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def analyze_ticket_urgency
    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 eksempel initialiseres `SupportTicketScheduler` med et supportanmodningsobjekt og vedligeholder en planlægningslog. AI-komponenten analyserer anmodningens detaljer og planlægger forudsigende opgavetildelingen baseret på faktorer som anmodningens hastende karakter, medarbejderkompetence og forventet medarbejderarbejdsbelastning.

De tilgængelige funktioner, som AI'en kan anvende, omfatter `analyze_ticket_urgency`, `list_available_agents`, `predict_agent_workload` og `assign_ticket_to_agent`. Hver funktion delegerer opgaven til en tilsvarende analyse- eller prædiktionskomponent og tilføjer resultatet til planlægningsloggen. AI'en har også mulighed for at udsætte tildelingen ved hjælp af `delay_assignment`-funktionen.

AI-komponenten undersøger planlægningsloggen og træffer velinformerede beslutninger om opgavetildeling. Den tager højde for anmodningens hastende karakter, de tilgængelige medarbejderes kompetencer og den forventede arbejdsbelastning for hver medarbejder for at bestemme den mest egnede medarbejder til at håndtere opgaven.

Ved at udnytte forudsigende opgaveplanlægning kan kundesupportapplikationen optimere opgavetildeling, reducere svartider og forbedre den generelle kundetilfredshed. Proaktiv og effektiv håndtering af supportanmodninger sikrer, at de rigtige opgaver tildeles de rigtige medarbejdere på det rigtige tidspunkt.

## Fejlhåndtering og Genopretning i en Databehandlingspipeline

Håndtering af fejl og genopretning efter nedbrud er afgørende for at sikre dataintegritet og forhindre tab af data. `DataProcessingOrchestrator`-komponenten bruger AI til intelligent at håndtere fejl og orchestrere genopretningsprocessen i en databehandlingspipeline

```ruby
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 eksempel initialiseres `DataProcessingOrchestrator` med et databatch-objekt og vedligeholder en behandlingsprotokol. AI-komponenten orchestrerer databehandlingspipelinen, håndterer undtagelser og genopretter fra fejl efter behov.

De tilgængelige funktioner, som AI'en kan kalde, omfatter `validate_data`, `process_data`, `request_fix`, `retry_processing` og `mark_data_as_failed`. Hver funktion delegerer opgaven til en tilsvarende databehandlingskomponent og tilføjer resultatet eller undtagelsesdetaljerne til behandlingsprotokollen.

Hvis der opstår en valideringsundtagelse under `validate_data`-trinnet, tilføjer `handle_validation_exception`-funktionen undtagelsesdataene til protokollen og returnerer kontrollen til AI'en. Tilsvarende, hvis der opstår en behandlingsundtagelse under `process_data`-trinnet, kan AI'en beslutte genopretningsstrategien.

Afhængigt af den opståede undtagelses art kan AI'en efter eget skøn beslutte at kalde `request_fix`, som delegerer til en AI-drevet `SmartDataFixer`-komponent (se kapitlet om Selvhelende Data). Datafixeren får en almindelig dansk beskrivelse af,

hvordan den skal modificere `@data_batch`, så behandlingen kan genoptages. Måske ville en vellykket genoptagelse indebære at fjerne poster fra databatchen, som ikke har bestået valideringen og/eller kopiere dem til en anden behandlingspipeline til manuel gennemgang? Mulighederne er næsten uendelige.

Ved at inkorporere AI-drevet undtagelseshåndtering og genopretning bliver databehandlingsapplikationen mere modstandsdygtig og fejltolerant. `DataProcessingOrchestrator` håndterer intelligent undtagelser, minimerer datatab og sikrer en problemfri udførelse af databehandlingsarbejdsgangen.

# Overvågning og Logføring

Overvågning og logføring giver indblik i fremskridt, ydeevne og sundhed af AI-drevne arbejdsgangskomponenter, hvilket gør det muligt for udviklere at spore og analysere systemets adfærd. Implementering af effektive overvågnings- og logføringsmekanismer er afgørende for fejlfinding, revision og kontinuerlig forbedring af intelligente arbejdsgange.

## Overvågning af Arbejdsgangens Fremskridt og Ydeevne

For at sikre en problemfri udførelse af intelligente arbejdsgange er det vigtigt at overvåge fremskridt og ydeevne for hver arbejdsgangskomponent. Dette indebærer at spore nøgletal og begivenheder gennem arbejdsgangens livscyklus.

Vigtige aspekter at overvåge omfatter:

1. **Arbejdsgangens Udførelsestid**: Mål den tid, hver arbejdsgangskomponent bruger på at fuldføre sin opgave. Dette hjælper med at identificere flaskehalse i ydeevnen og optimere den samlede arbejdsgangseffektivitet.

2. **Ressourceforbrug**: Overvåg forbruget af systemressourcer, såsom CPU, hukommelse og lagerplads, for hver arbejdsgangskomponent. Dette hjælper med at sikre, at systemet opererer inden for sin kapacitet og effektivt kan håndtere arbejdsbyrden.

3. **Fejlrater og Undtagelser**: Spor forekomsten af fejl og undtagelser inden for arbejdsgangskomponenter. Dette hjælper med at identificere potentielle problemer og muliggør proaktiv fejlhåndtering og genopretning.

4. **Beslutningspunkter og Resultater**: Overvåg beslutningspunkterne i arbejdsgangen og resultaterne af AI-drevne beslutninger. Dette giver indsigt i AI-komponenternes adfærd og effektivitet.

De data, der opfanges af overvågningsprocesser, kan vises i dashboards eller bruges som input til planlagte rapporter, der informerer systemadministratorer om systemets sundhed.

Overvågningsdata kan fødes til en AI-drevet systemadministratorproces til gennemgang og potentiel handling!

## Logføring af Vigtige Begivenheder og Beslutninger

Logføring er en essentiel praksis, der involverer opfangning og lagring af relevant information om vigtige begivenheder, beslutninger og undtagelser, der opstår under arbejdsgangens udførelse.

Vigtige aspekter at logføre omfatter:

1. **Arbejdsgangens Initiering og Fuldførelse**: Log start- og sluttidspunkter for hver arbejdsgangsinstans, sammen med relevant metadata såsom inputdata og brugerkontekst.

2. **Komponentudførelse**: Log udførelsesdetaljerne for hver arbejdsgangskomponent, herunder inputparametre, outputresultater og eventuelle mellemliggende data, der genereres.

3. **AI-beslutninger og Ræsonnement**: Log de beslutninger, der træffes af AI-komponenter, sammen med den underliggende begrundelse eller konfidensscorer. Dette giver gennemsigtighed og muliggør revision af AI-drevne beslutninger.

4. **Undtagelser og Fejlmeddelelser**: Log eventuelle undtagelser eller fejlmeddelelser, der opstår under arbejdsgangens udførelse, herunder staksporing og relevant kontekstinformation.

Logføring kan implementeres ved hjælp af forskellige teknikker, såsom at skrive til logfiler, gemme logs i en database eller sende logs til en centraliseret logføringstjeneste. Det er vigtigt at vælge et logføringsframework, der giver fleksibilitet, skalerbarhed og nem integration med applikationens arkitektur.

Her er et eksempel på, hvordan logføring kan implementeres i en Ruby on Rails-applikation ved hjælp af `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 strategisk at placere logningserklæringer gennem arbejdsgangskomponenterne og AI-beslutningspunkterne kan udviklere indfange værdifuld information til fejlfinding, revision og analyse.

## Fordele ved Overvågning og Logning

Implementering af overvågning og logning i intelligent arbejdsgangsorkestrering giver flere fordele:

1. **Fejlfinding og Fejlsøgning**: Detaljerede logs og overvågningsdata hjælper udviklere med hurtigt at identificere og diagnosticere problemer. De giver indsigt i arbejdsgangens udførelsesflow, komponentinteraktioner og eventuelle fejl eller undtagelser, der opstår.

2. **Ydeevneoptimering**: Overvågning af ydeevnemetrikker giver udviklere mulighed for at identificere flaskehalse og optimere arbejdsgangskomponenterne for bedre effektivitet. Ved at analysere udførelsestider, ressourceforbrug og andre metrikker kan udviklere træffe informerede beslutninger for at forbedre systemets overordnede ydeevne.

3. **Revision og Overholdelse**: Logning af vigtige hændelser og beslutninger giver et revisionsspor for regulatorisk overholdelse og ansvarlighed. Det gør det muligt for organisationer at spore og verificere de handlinger, der udføres af AI-komponenter og sikre overholdelse af forretningsregler og lovkrav.

4. **Kontinuerlig Forbedring**: Overvågnings- og logningsdata fungerer som værdifulde input til kontinuerlig forbedring af intelligente arbejdsgange. Ved at analysere historiske data, identificere mønstre og måle effektiviteten af AI-beslutninger kan udviklere iterativt forfine og forbedre arbejdsgangsorkestreringslogikken.

## Overvejelser og Bedste Praksis

Ved implementering af overvågning og logning i intelligent arbejdsgangsorkestrering bør følgende bedste praksis overvejes:

1. **Definer Klare Overvågningsmetrikker**: Identificer de vigtigste metrikker og hændelser, der skal overvåges baseret på arbejdsgangens specifikke krav. Fokuser på metrikker, der giver meningsfuld indsigt i systemets ydeevne, sundhed og adfærd.

2. **Implementer Detaljeret Logning**: Sørg for at logningserklæringer er placeret på passende steder inden for arbejdsgangskomponenterne og AI-beslutningspunkterne. Indfang relevant kontekstinformation, såsom inputparametre, outputresultater og eventuelle mellemliggende data, der genereres.

3. **Brug Struktureret Logning**: Anvend et struktureret logningsformat for at lette nem parsing og analyse af logdata. Struktureret logning muliggør bedre søgbarhed, filtrering og aggregering af logposter.

4. **Administrer Logopbevaring og -rotation**: Implementer politikker for logopbevaring og -rotation for at administrere lagring og livscyklus af logfiler. Fastlæg den passende opbevaringsperiode baseret på lovkrav, lagringsbegrænsninger og analysebehov. Hvis muligt, udliciter logning til en tredjepartstjeneste som Papertrail.

5. **Sikr Følsom Information**: Vær forsigtig ved logning af følsom information, såsom personhenførbare oplysninger (PII) eller fortrolige forretningsdata. Implementer passende sikkerhedsforanstaltninger, såsom datamaskning eller kryptering, for at beskytte følsom information i logfiler.

6. **Integrer med Overvågnings- og Alarmeringsværktøjer**: Udnyt overvågnings- og alarmeringsværktøjer til at centralisere indsamling, analyse og visualisering af overvågnings- og logningsdata. Disse værktøjer kan give realtidsindsigt, generere alarmer baseret på foruddefinerede tærskler og lette proaktiv problemdetektion og -løsning. Mit foretrukne af disse værktøjer er Datadog.

Ved at implementere omfattende overvågnings- og logningsmekanismer kan udviklere opnå værdifuld indsigt i adfærden og ydeevnen af intelligente arbejdsgange. Disse indsigter muliggør effektiv fejlfinding, optimering og kontinuerlig forbedring af AI-drevne arbejdsgangsorkestreringsystemer.

## Skalerbarhed og Ydeevneovervejelser

Skalerbarhed og ydeevne er kritiske aspekter at overveje ved design og implementering af intelligente arbejdsgangsorkestreringsystemer. Efterhånden som mængden af samtidige arbejdsgange og kompleksiteten af AI-drevne komponenter øges, bliver det essentielt at sikre, at systemet kan håndtere arbejdsbyrden effektivt og skalere problemfrit for at imødekomme voksende krav.

## Håndtering af Store Mængder Samtidige Arbejdsgange

Intelligente arbejdsgangsorkestreringsystemer skal ofte håndtere et stort antal samtidige arbejdsgange. For at sikre skalerbarhed bør følgende strategier overvejes:

1. **Asynkron Behandling**: Implementer asynkrone behandlingsmekanismer for at afkoble udførelsen af arbejdsgangskomponenter. Dette gør det muligt for systemet at håndtere flere arbejdsgange samtidigt uden at blokere eller vente på, at hver komponent færdiggøres. Asynkron behandling kan opnås ved hjælp af meddelelseskøer, hændelsesdrevne arkitekturer eller baggrundsjobbehandlingsframeworks som Sidekiq.

2. **Distribueret Arkitektur**: Design systemarkitekturen til at bruge serverløse komponenter (såsom AWS Lambda) eller simpelthen distribuere arbejdsbyrden på tværs af flere noder eller servere sammen med din hovedapplikationsserver. Dette muliggør horisontal skalerbarhed, hvor yderligere noder kan tilføjes for at håndtere øgede arbejdsgangsmængder.

3. **Parallel Udførelse**: Identificer muligheder for parallel udførelse inden for arbejdsgange. Nogle arbejdsgangskomponenter kan være uafhængige af hinanden og kan udføres samtidigt. Ved at udnytte parallelle behandlingsteknikker, såsom multi-threading eller distribuerede opgavekøer, kan systemet optimere ressourceudnyttelsen og reducere den samlede arbejdsgangsudførelsestid.

## Optimering af Ydeevne for AI-drevne Komponenter

AI-drevne komponenter, såsom maskinlæringsmodeller eller sprogbehandlingsmotorer, kan være beregningstunge og påvirke den overordnede ydeevne af arbejdsprocesstyringssystemet. For at optimere ydeevnen af AI-komponenter bør følgende teknikker overvejes:

1. **Caching**: Hvis din AI-behandling er rent generativ og ikke involverer realtidsopslag eller eksterne integrationer for at generere chat-fuldførelser, kan du undersøge

cachingmekanismer til at gemme og genbruge resultater fra hyppigt anvendte eller beregningstunge operationer.

2. **Modeloptimering**: Optimer løbende den måde, du bruger AI-modeller i arbejdsprocesskomponenter. Dette kan involvere teknikker som *Prompt-destillation* eller det kan simpelthen være et spørgsmål om at teste nye modeller, efterhånden som de bliver tilgængelige.

3. **Batchbehandling**: Hvis du arbejder med GPT-4-klasse modeller, kan du muligvis udnytte batchbehandlingsteknikker til at behandle flere datapunkter eller forespørgsler i en enkelt batch i stedet for at behandle dem individuelt. Ved at behandle data i batches kan systemet optimere ressourceudnyttelsen og reducere overheaden fra gentagne modelforespørgsler.

## Overvågning og Profilering af Ydeevne

For at identificere flaskehalse i ydeevnen og optimere skalerbarheden af det intelligente arbejdsprocesstyringssystem, er det afgørende at implementere overvågnings- og profileringsmekanismer. Overvej følgende tilgange:

1. **Ydelsesmålinger**: Definer og spor centrale ydelsesmålinger, såsom svartid, gennemløb, ressourceudnyttelse og latenstid. Disse målinger giver indsigt i systemets ydeevne og hjælper med at identificere områder til optimering. Den populære AI-model-aggregator OpenRouter inkluderer Host[1]- og Speed[2]-målinger i hvert API-svar, hvilket gør det enkelt at spore disse centrale målinger.

2. **Profileringsværktøjer**: Brug profileringsværktøjer til at analysere ydeevnen af individuelle arbejdsprocesskomponenter og AI-operationer. Profileringsværktøjer kan hjælpe med at identificere ydeevnehotspots, ineffektive kodestier eller ressourcekrævende operationer. Populære profileringsværktøjer omfatter New

[1]Host er den tid, det tog at modtage den første byte af den streamede generering fra modelværten, også kendt som "time to first byte."

[2]Speed beregnes som antallet af fuldførelsestokens divideret med den samlede genereringstid. For ikke-streamede forespørgsler betragtes latenstid som en del af genereringstiden.

Relic, Scout eller indbyggede profileringsværktøjer fra programmeringssproget eller framework'et.

3. **Belastningstest**: Udfør belastningstest for at evaluere systemets ydeevne under forskellige niveauer af samtidige arbejdsbelastninger. Belastningstest hjælper med at identificere systemets skalerbarhedsgrænser, opdage forringelse af ydeevnen og sikre, at systemet kan håndtere den forventede trafik uden at kompromittere ydeevnen.

4. **Kontinuerlig Overvågning**: Implementer kontinuerlig overvågning og alarmeringsmekanismer for proaktivt at opdage ydelsesproblemer og flaskehalse. Opsæt overvågningsdashboards og alarmer til at spore centrale præstationsindikatorer (KPI'er) og modtag notifikationer, når foruddefinerede grænseværdier overskrides. Dette muliggør hurtig identifikation og løsning af ydelsesproblemer.

## Skaleringsstrategier

For at håndtere stigende arbejdsbelastninger og sikre skalerbarheden af det intelligente arbejdsprocesstyringssystem, bør følgende skaleringsstrategier overvejes:

1. **Vertikal Skalering**: Vertikal skalering involverer forøgelse af ressourcerne (f.eks. CPU, hukommelse) på individuelle noder eller servere for at håndtere højere arbejdsbelastninger. Denne tilgang er velegnet, når systemet kræver mere processorkraft eller hukommelse til at håndtere komplekse arbejdsprocesser eller AI-operationer.

2. **Horisontal Skalering**: Horisontal skalering involverer tilføjelse af flere noder eller servere til systemet for at fordele arbejdsbelastningen. Denne tilgang er effektiv, når systemet skal håndtere et stort antal samtidige arbejdsprocesser, eller når arbejdsbelastningen nemt kan fordeles på tværs af flere noder. Horisontal skalering kræver en distribueret arkitektur og load balancing-mekanismer for at sikre jævn fordeling af trafikken.

3. **Auto-skalering**: Implementer auto-skaleringsmekanismer til automatisk at justere antallet af noder eller ressourcer baseret på arbejdsbelastningsbehovet. Auto-skalering

tillader systemet dynamisk at skalere op eller ned afhængigt af den indkommende trafik, hvilket sikrer optimal ressourceudnyttelse og omkostningseffektivitet. Cloudplatforme som Amazon Web Services (AWS) eller Google Cloud Platform (GCP) tilbyder auto-skaleringsmuligheder, der kan udnyttes til intelligente arbejdsprocesstyringssystemer.

## Ydeevneoptimeringteknikker

Ud over skaleringsstrategierne bør følgende ydeevneoptimeringteknikker overvejes for at forbedre effektiviteten af det intelligente arbejdsprocesstyringssystem:

1. **Effektiv Datalagring og -hentning**: Optimer de datalagring- og hentningsmekanismer, der bruges af arbejdsprocesskomponenterne. Brug effektiv databaseindeksering, forespørgselsoptimeringteknikker og datacaching for at minimere latenstiden og forbedre ydeevnen af dataintensive operationer.

2. **Asynkron I/O**: Brug asynkrone I/O-operationer for at forhindre blokering og forbedre systemets reaktionsevne. Asynkron I/O gør det muligt for systemet at håndtere flere anmodninger samtidigt uden at vente på, at I/O-operationer bliver færdige, hvorved ressourceudnyttelsen maksimeres.

3. **Effektiv serialisering og deserialisering**: Optimér de serialiserings- og deserialiseringsprocesser, der bruges til dataudveksling mellem workflow-komponenter. Brug effektive serialiseringsformater som Protocol Buffers eller MessagePack for at reducere overhead ved dataserialisering og forbedre ydeevnen af kommunikationen mellem komponenter.

For Ruby-baserede applikationer kan du overveje at bruge Universal ID. Universal ID udnytter både MessagePack og Brotli (en kombination bygget til hastighed og førsteklasses datakomprimering). Når disse biblioteker kombineres, er de op til 30% hurtigere og inden for 2-5% komprimeringsrater sammenlignet med Protocol Buffers.

4. **Komprimering og kodning**: Anvend kompressions- og kodningsteknikker for at reducere størrelsen af data, der overføres mellem workflow-komponenter. Komprimeringsalgoritmer som gzip eller Brotli kan markant reducere netværksbåndbreddeforbruget og forbedre systemets samlede ydeevne.

Ved at tage hensyn til skalerbarhed og ydeevneaspekter under design og implementering af intelligente workflow-orchestreringssystemer, kan du sikre, at dit system kan håndtere store mængder samtidige workflows, optimere ydeevnen af AI-drevne komponenter og skalere problemfrit for at imødekomme voksende krav. Kontinuerlig overvågning, profilering og optimering er afgørende for at opretholde systemets ydeevne og reaktionsevne, efterhånden som arbejdsbelastningen og kompleksiteten øges over tid.

# Test og validering af workflows

Test og validering er kritiske aspekter af udvikling og vedligeholdelse af intelligente workflow-orchestreringssystemer. I betragtning af den komplekse natur af AI-drevne workflows er det afgørende at sikre, at hver komponent fungerer som forventet, at det overordnede workflow opfører sig korrekt, og at AI-beslutningerne er nøjagtige og pålidelige. I dette afsnit vil vi udforske forskellige teknikker og overvejelser for test og validering af intelligente workflows.

## Enhedstest af workflow-komponenter

Enhedstest involverer test af individuelle workflow-komponenter isoleret for at verificere deres korrekthed og robusthed. Når du udfører enhedstest af AI-drevne workflow-komponenter, bør du overveje følgende:

1. **Input-validering**: Test komponentens evne til at håndtere forskellige typer input, herunder gyldige og ugyldige data. Verificér at komponenten håndterer grænsetilfælde elegant og giver passende fejlmeddelelser eller undtagelser.

2. **Output-verifikation**: Bekræft at komponenten producerer det forventede output for et givent sæt inputs. Sammenlign det faktiske output med de forventede resultater for at sikre korrekthed.

3. **Fejlhåndtering**: Test komponentens fejlhåndteringsmekanismer ved at simulere forskellige fejlscenarier, såsom ugyldigt input, utilgængelige ressourcer eller uventede undtagelser. Verificér at komponenten fanger og håndterer fejl korrekt.

4. **Grænseværdier**: Test komponentens opførsel under grænseværdibetingelser, såsom tomt input, maksimal inputstørrelse eller ekstreme værdier. Sikr at komponenten håndterer disse betingelser elegant uden at gå ned eller producere ukorrekte resultater.

Her er et eksempel på en enhedstest for en workflow-komponent i Ruby ved hjælp af RSpec test-frameworket:

```
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 eksempel testes `OrderValidator`-komponenten ved hjælp af to testtilfælde: ét for en gyldig ordre og et andet for en ugyldig ordre. Testtilfældene verificerer,

at `validate`-metoden returnerer den forventede booleske værdi baseret på ordrens gyldighed.

## Integration Testing af Arbejdsgangsinteraktioner

Integrationstest fokuserer på at verificere interaktioner og dataflow mellem forskellige arbejdsgangskomponenter. Det sikrer, at komponenterne arbejder problemfrit sammen og producerer de forventede resultater. Når der udføres integrationstest af intelligente arbejdsgange, bør følgende overvejes:

1. **Komponentinteraktion**: Test kommunikationen og dataudvekslingen mellem arbejdsgangskomponenter. Verificér at outputtet fra én komponent korrekt videregives som input til den næste komponent i arbejdsgangen.

2. **Datakonsistens**: Sikr at data forbliver konsistent og præcis, mens det flyder gennem arbejdsgangen. Verificér at datatransformationer, beregninger og aggregeringer udføres korrekt.

3. **Undtagelseshåndtering**: Test hvordan undtagelser og fejl forplanter sig og håndteres på tværs af arbejdsgangskomponenter. Verificér at undtagelser opfanges, logges og håndteres hensigtsmæssigt for at forhindre forstyrrelser i arbejdsgangen.

4. **Asynkron Adfærd**: Hvis arbejdsgangen involverer asynkrone komponenter eller parallel eksekvering, test da koordinerings- og synkroniseringsmekanismerne. Sikr at arbejdsgangen opfører sig korrekt under samtidige og asynkrone scenarier.

Her er et eksempel på en integrationstest for en arbejdsgang i Ruby ved hjælp af RSpec test-frameworket:

```
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 eksempel testes `OrderProcessingWorkflow` ved at verificere samspillet mellem forskellige arbejdsgangskomponenter. Testscenariet opstiller forventninger til hver komponents adfærd og sikrer, at arbejdsgangen behandler ordren succesfuldt og opdaterer ordrens status i overensstemmelse hermed.

## Test af AI-beslutningspunkter

Test af AI-beslutningspunkter er afgørende for at sikre nøjagtigheden og pålideligheden af AI-drevne arbejdsgange. Ved test af AI-beslutningspunkter bør man overveje følgende:

1. **Beslutningsnøjagtighed**: Verificér at AI-komponenten træffer nøjagtige beslutninger baseret på inputdata og den trænede model. Sammenlign AI-beslutningerne med forventede resultater eller referencedata.

2. **Grænsetilfælde**: Test AI-komponentens adfærd under grænsetilfælde og usædvanlige scenarier. Verificér at AI-komponenten håndterer disse tilfælde elegant og træffer fornuftige beslutninger.

3. **Bias og retfærdighed**: Vurdér AI-komponenten for potentielle bias og sikr, at den træffer fair og upartiske beslutninger. Test komponenten med forskelligartede inputdata og analysér resultaterne for eventuelle diskriminerende mønstre.

4. **Forklarlighed**: Hvis AI-komponenten giver forklaringer eller begrundelser for sine beslutninger, skal du verificere, at forklaringerne er korrekte og klare. Sikr at forklaringerne stemmer overens med den underliggende beslutningsproces.

Her er et eksempel på test af et AI-beslutningspunkt i Ruby ved brug af RSpec-testrammeværket:

```ruby
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 eksempel testes `FraudDetector` AI-komponenten med to testtilfælde: et for en svigagtig transaktion og et andet for en legitim transaktion. Testtilfældene verificerer, at `detect_fraud`-metoden returnerer den forventede booleske værdi baseret på transaktionens karakteristika.

## End-to-End Test

End-to-end test involverer test af hele arbejdsgangen fra start til slut, hvor man simulerer virkelige scenarier og brugerinteraktioner. Det sikrer, at arbejdsgangen opfører sig korrekt og producerer de ønskede resultater. Når der udføres end-to-end test af intelligente arbejdsgange, bør man overveje følgende:

1. **Brugerscenarier**: Identificér almindelige brugerscenarier og test arbejdsgangens adfærd under disse scenarier. Verificér at arbejdsgangen håndterer brugerinput korrekt, træffer passende beslutninger og producerer de forventede output.

2. **Datavalidering**: Sikr at arbejdsgangen validerer og behandler brugerinput for at forhindre datauoverensstemmelser eller sikkerhedssårbarheder. Test arbejdsgangen med forskellige typer inputdata, herunder både gyldige og ugyldige data.

3. **Fejlhåndtering**: Test arbejdsgangens evne til at komme sig efter fejl og undtagelser. Simulér fejlscenarier og verificér, at arbejdsgangen håndterer dem elegant, logger fejlene og udfører passende genopretningshandlinger.

4. **Ydeevne og Skalerbarhed**: Vurdér arbejdsgangens ydeevne og skalerbarhed under forskellige belastningsforhold. Test arbejdsgangen med en stor mængde samtidige anmodninger og mål responstider, ressourceforbrug og systemets generelle stabilitet.

Her er et eksempel på en end-to-end test af en arbejdsgang i Ruby ved hjælp af RSpec testframework og Capybara-biblioteket til simulering af brugerinteraktioner:

```
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 eksempel simulerer end-to-end testen en bruger, der afgiver en ordre gennem webgrænsefladen. Den udfylder de påkrævede formularfelter, indsender ordren og verificerer, at ordren behandles korrekt, viser den passende bekræftelsesbesked og opdaterer ordrestatus i databasen.

## Kontinuerlig Integration og Deployment

For at sikre pålideligheden og vedligeholdelsen af intelligente arbejdsgange anbefales det at integrere test og validering i den kontinuerlige integrations- og deployment (CI/CD) pipeline. Dette muliggør automatiseret test og validering af ændringer i arbejdsgangen, før de implementeres i produktion. Overvej følgende praksisser:

1. **Automatiseret Testkørsel**: Konfigurer CI/CD-pipelinen til automatisk at køre test-suiten, når der foretages ændringer i arbejdsgangens kodebase. Dette sikrer, at eventuelle regressioner eller fejl opdages tidligt i udviklingsprocessen.

2. **Overvågning af Testdækning**: Mål og overvåg testdækningen af arbejdsgangens komponenter og AI-beslutningspunkter. Stræb efter høj testdækning for at sikre, at kritiske stier og scenarier testes grundigt.

3. **Løbende Feedback**: Integrer testresultater og kodekvalitetsmetrikker i udviklingsarbejdsgangen. Giv løbende feedback til udviklere om testenes status,

kodekvalitet og eventuelle problemer, der opdages under CI/CD-processen.

4. **Staging-miljøer**: Implementer arbejdsgangen i staging-miljøer, der nøje afspejler produktionsmiljøet. Udfør yderligere test og validering i staging-miljøet for at fange eventuelle problemer relateret til infrastruktur, konfiguration eller dataintegration.

5. **Rollback-mekanismer**: Implementer rollback-mekanismer i tilfælde af implementeringsfejl eller kritiske problemer opdaget i produktion. Sørg for, at arbejdsgangen hurtigt kan rulles tilbage til en tidligere stabil version for at minimere nedetid og påvirkning af brugerne.

Ved at inkorporere test og validering gennem hele udviklingslivscyklussen for intelligente arbejdsgange kan organisationer sikre pålideligheden, nøjagtigheden og vedligeholdelsen af deres AI-drevne systemer. Regelmæssig test og validering hjælper med at fange fejl, forebygge regressioner og opbygge tillid til arbejdsgangens adfærd og resultater.

# Del 2: Mønstrene

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Prompt Engineering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Chain of Thought

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan det virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempler

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Indholdsproduktion

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Struktureret Entitetsoprettelse

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Vejledning af LLM-agenter

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Fordele og Overvejelser

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Tilstandsskift

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan Fungerer Det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvornår skal det bruges

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Rolletildeling

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan det virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvornår skal det bruges

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempler

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Prompt-objekt

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan fungerer det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Promptskabelon

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan fungerer det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Fordele og overvejelser

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvornår skal det bruges:

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Structured IO

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan det virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Skalering af Struktureret IO

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Fordele og overvejelser

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Prompt-kædekobling

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan det virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvornår det skal bruges

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel: Olympias Onboarding

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Prompt-omskriver

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan fungerer det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Response Fencing

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan det virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Fordele og Overvejelser

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Fejlhåndtering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Forespørgselsanalysator

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan det virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Implementering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Ordklassemærkning (POS) og Navngiven Entitetsgenkendelse (NER)

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Intentionsklassificering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Nøgleordsudtrækning

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Fordele

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Forespørgselsomskriver

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan virker det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Fordele

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Ventriloquist

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan det virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvornår skal det bruges

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Diskrete Komponenter

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Prædikat

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan Det Virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvornår skal det bruges

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# API-facade

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan det virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Centrale fordele

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvornår det skal bruges

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Autentificering og Autorisering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Håndtering af Forespørgsler

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Formatering af Svar

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Fejlhåndtering og Særtilfælde

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Overvejelser om Skalerbarhed og Ydeevne

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sammenligning med Andre Designmønstre

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Resultatfortolker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan fungerer det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvornår skal det bruges

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Virtuel Maskine

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan Det Virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvornår Det Skal Bruges

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Bag Magien

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Specifikation og Test

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Specificering af Adfærd

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Skrivning af Testtilfælde

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel: Test af Oversætterkomponenten

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Genafspilning af HTTP-interaktioner

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Human In The Loop (HITL)

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Overordnede Mønstre

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Hybrid Intelligens

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Adaptiv Respons

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Menneske-AI-rolleskift

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Eskalering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan fungerer det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Vigtige fordele

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Anvendelse i Praksis: Sundhedsvæsenet

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Feedbacksløjfe

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan Det Virker

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Anvendelser og Eksempler

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Avancerede Teknikker i Integration af Menneskelig Feedback

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Passiv Informationsudstråling

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan Det Fungerer

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Kontekstuel Informationsvisning

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Proaktive Notifikationer

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Forklarende Indsigt

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Interaktiv Udforskning

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Centrale Fordele

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Anvendelser og Eksempler

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Kollaborativ Beslutningstagning (CDM)

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan Det Fungerer

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Kontinuerlig Læring

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan Fungerer Det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Anvendelser og Eksempler

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Etiske Overvejelser

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## HITL's Rolle i Reduktion af AI-Risici

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Teknologiske Fremskridt og Fremtidsudsigter

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Udfordringer og Begrænsninger ved HITL-Systemer

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Intelligent Fejlhåndtering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Traditionelle Fejlhåndteringstilgange

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Kontekstuel fejldiagnose

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan fungerer det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Prompt-engineering til kontekstuel fejldiagnose

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### Retrieval-Augmented Generation til kontekstuel fejldiagnose

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Intelligent fejlrapportering

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Forebyggende Fejlprævention

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan Fungerer Det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Intelligent Fejlgenopretning

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan Fungerer Det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Personaliseret Fejlkommunikation

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan Fungerer Det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Adaptiv Fejlhåndteringsarbejdsgang

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan fungerer det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Kvalitetskontrol

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Eval

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Problem

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Løsning

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan fungerer det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Overvejelser

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Forståelse af Gyldne Referencer

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Hvordan Referencefri Evalueringer Fungerer

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Sikkerhedsmekanisme

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Problem

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Løsning

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Sådan fungerer det

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Eksempel

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Overvejelser

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Sikkerhedsforanstaltninger og Evalueringer: To Sider af Samme Sag

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Udskifteligheden mellem Sikkerhedsforanstaltninger og Referencefri Evalueringer

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Implementering af Tosidede Sikkerhedsforanstaltninger og Evalueringer

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Ordliste

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Ordliste

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### A

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### B

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

### C

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## D

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## E

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## F

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## G

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## H

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## I

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## J

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## K

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## L

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## M

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## N

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## O

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## P

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Q

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## R

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## S

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## T

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## U

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## V

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## W

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

## Z

Dette indhold er ikke tilgængeligt i prøvebogen. Bogen kan købes på Leanpub på http://leanpub.com/patterns-of-application-development-using-ai-da.

# Index