UPPSALA
UNIVERSITET

UPTEC STS 26033
Examensarbete 30 hp
Juni 2026

Bortom traditionella kodmått

Ett kontextkänsligt ramverk för kvalitetsutvärdering av AI-genererad kod

Sardal, Ingrid
Westin, David

Civilingenjörsprogrammet i system i teknik och samhälle

UPPSALA
UNIVERSITET

Beyond Traditional Code Metrics
A Context-Aware Framework for Quality Evaluation of AI-Generated Code

Sardal, Ingrid
Westin, David

Abstract

The increasing use of generative AI in software development, particularly through large language models, has changed how code is produced and raised new questions regarding how the quality of such code should be assessed. Traditional algorithmic code quality metrics were developed for human-written code at a time when codebases were significantly smaller and simpler than today. Therefore, these metrics may not fully capture the characteristics of AI-generated code in modern software systems. This thesis investigates how AI-generated code can be evaluated in a reliable and meaningful way, focusing on algorithmic metrics as well as specification compliance and contextual interpretation. The study combines a literature review, interviews with developers, and an empirical analysis of AI-generated code. Based on these, an evaluation framework was developed that integrates deterministic algorithmic metrics with an AI-based reviewer, which interpreted the results in relation to the generated code's functional purpose and broader system context using relevant repository files.

The framework was validated by comparing its reviews with manual code reviews. The results show that developers are still better at identifying subtle semantic issues and design-related considerations, even though the framework catches some of these aspects. In contrast, the framework is more effective at detecting structural code properties. The findings highlight a growing need for automated evaluation tools as the scale of AI-generated code makes manual review increasingly impractical. The thesis concludes that reliable evaluation of AI-generated code requires a hybrid approach, where deterministic metrics provide a stable foundation and AI contributes contextual interpretation.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Lukas Bergliden Ämnesgranskare: Carl Nettelblad

Examinator: Elísabet Andrésdóttir

Sammanfattning

Inom mjukvaruutveckling har generativ AI snabbt gått från att vara ett experimentellt hjälpmedel till att bli en självklar del av utvecklarnas vardag. Stora språkmodeller, så kallade LLM:er, som ChatGPT, Gemini och Claude används för att skriva kod, föreslå lösningar och generera tester, vilket medför att stora mängder kod kan produceras på betydligt kortare tid än tidigare. Samtidigt väcker denna utveckling grundläggande frågor om kodens kvalitet. När kod genereras automatiskt och i hög volym blir det avgörande att kunna bedöma och följa om koden faktiskt är välskriven, underhållbar och anpassad till sitt sammanhang.

Traditionellt har kodkvalitet bedömts genom en kombination av algoritmiska mått, testtäckning och manuell granskning från utvecklare. Algoritmiska mått som cyklomatisk komplexitet, underhållbarhetsindex och sammanhållning har använts inom industrin i flera decennier och är fortfarande väletablerade. Dessa mått utvecklades dock i en tid då kodbaser såg fundamentalt annorlunda ut än idag, och tidigare forskning har visat att de inte alltid är tillräckliga för att fånga vad utvecklare faktiskt upplever som god kodkvalitet. Frågan om hur kvaliteten hos AI-genererad kod ska utvärderas är därför ett område där det idag saknas tydlig konsensus.

Denna uppsats har undersökt hur kvaliteten hos AI-genererad kod kan utvärderas på ett tillförlitligt och meningsfullt sätt. Studien har särskilt fokuserat på att jämföra traditionella algoritmiska kodutvärderingsmått med utvärdering baserad på kravuppfyllelse, kontext och mänsklig bedömning. Arbetet utfördes i samarbete med IT-konsultföretaget Decerno och bestod av fem faser: en litteraturstudie, en intervjustudie, en empirisk analys, utformning av ett eget kodutvärderingsverktyg, samt validering av verktyget.

Det utvecklade verktyget benämns i denna uppsats som utvärderingsramverket och kombinerar traditionella algoritmiska mått med en AI-reviewer. De algoritmiska måtten som inkluderades var cyklomatisk komplexitet, underhållbarhetsindex, sammanhållningsmåtten TCC och LCOM1 samt kodduplicering. Dessa mått fungerar tillsammans som ett stabilt underlag som AI-reviewern sedan tolkar i förhållande till kodens specifika uppgift, design och roll i den övergripande systemkontexten. På så sätt kompletteras de strukturella mätvärdena med en mer nyanserad och kontextkänslig bedömning, vilket bättre speglar hur mänskliga utvecklare själva granskar kod. För att stärka tillförlitligheten i AI-reviewerns bedömningar tillämpades dessutom en tvåstegsmetod för prompting, där modellen först tvingas att genomföra en strukturerad analys innan den fattar ett beslut.

För att undersöka hur väl ramverket fångar kodkvalitet validerades det mot manuell kodgranskning utförd av tre utvecklare med olika lång arbetslivserfarenhet. Fyra AI-genererade kodsegment från tre olika projekt granskades både av utvecklarna och av utvärderingsramverket, varpå resultaten jämfördes. Resultaten visade att ramverket och utvecklarna i hög grad delade synen på kodsegmentens kvalitet. Båda fokuserade bland annat på huruvida kodens funktionalitet motsvarade kravspecifikationen och båda upptäckte kodduplicering och hög cyklomatisk komplexitet. Däremot framkom det att utvecklarna i högre grad fångade subtila aspekter såsom semantiska val, designrelaterade

beslut och tillfälliga lösningar i koden, även om utvärderingsramverket till viss del också identifierade sådana inslag. Samtidigt visade sig utvärderingsramverket vara träffsäkert i bedömningen av kodens strukturella egenskaper.

Studien visar att traditionella algoritmiska kodutvärderingsmått, även om de fortfarande har ett värde, inte är tillräckliga som ensamt utvärderingsverktyg. Detta gäller oavsett om koden är AI-genererad eller skriven av människor. Måtten saknar förmåga att tolka kodens specifika uppgift, syfte och roll i en större systemkontext, och behöver därför kompletteras med en kontextkänslig bedömning. Vid utvärdering av specifikt AI-genererad kod blir behovet av ett automatiserat utvärderingsverktyg särskilt påtagligt. Detta eftersom AI-genererade kodbaser tenderar att bli mycket stora, vilket gör manuell granskning av all kod orealistisk. Utvecklare på företaget beskrev att förhållandet mellan kodutveckling och kodgranskning har förändrats i takt med att AI används för att generera kod, där granskning idag tar betydligt mer tid. I ett vidare perspektiv aktualiserar resultaten också frågan om hur framtidens granskningskompetens kommer att utvecklas. Om generativ AI tar en allt större roll i kodproduktionen kan möjligheterna att bygga den erfarenhetsbaserade och intuitiva förståelse som mänsklig kodgranskning ofta vilar på minska. Detta kan sin tur öka behovet av tillförlitliga, automatiserade utvärderingsverktyg.

Sammanfattningsvis pekar uppsatsens resultat på att tillförlitlig utvärdering av AI-genererad kod kräver en kombination av deterministiska algoritmiska mått och en AI-reviewer. De förstnämnda förankrar bedömningen, medan AI-reviewern bidrar med den kontextuella tolkning som krävs för att fånga kodens faktiska kvalitet i sitt sammanhang. Denna hybridlösning möjliggör en nyanserad bedömning, där de algoritmiska måtten ger en stabil grund medan AI-reviewern tolkar dessa i ljuset av kodens specifika uppgift, design och systemkontext. På så sätt kan utvärderingen fånga både kvantifierbara och svårfångade kvalitetsaspekter i ett och samma ramverk.

Förord

Vi vill rikta ett varmt tack till alla som på olika sätt har bidragit till genomförandet av detta examensarbete.

Först och främst vill vi tacka vår handledare Lukas Bergliden för värdefull vägledning, konstruktiv återkoppling och framförallt roliga och givande samtal. Vi vill också tacka Filippa Nilsson för att ha agerat bollplank under arbetets gång.

Vi vill vidare tacka vår ämnesgranskare Carl Nettelblad för viktiga synpunkter och perspektiv som har stärkt arbetets akademiska kvalitet och tydlighet, samt för att ha granskat arbetet med noggrannhet och engagemang.

Tack också till vår examinator och programansvarig Elísabet Andrésdóttir för långvarigt engagemang i våra studier.

Slutligen vill vi rikta ett stort tack till gänget i MigAIrate som lät oss vara en del av deras projekt under fem månader. Dessutom vill vi tacka alla de personer på Decerno som ställt upp på intervjuer och gjort studien möjlig genom att dela med sig av erfarenheter, kunskap och tid. Vi vill även tacka övriga medarbetare som välkomnat oss på Decerno och visat intresse för vårt arbete.

Uppsala, juni 2026

Ingrid Sardal & David Westin

Innehållsförteckning

1. Inledning

Under de senaste åren har artificiell intelligens fått en alltmer framträdande roll inom mjukvaruutveckling. Large Language Models (LLM) används idag för att skriva kod, föreslå lösningar, felsöka, generera tester och dokumentation samt stödja utvecklare i deras dagliga arbete. Verktyg som ChatGPT, GitHub Copilot och Claude används som en integrerad del av mjukvaruutvecklares vardag (Batte, 2025, Introduction). Denna utveckling innebär en betydande förändring av hur mjukvara produceras, både vad gäller hastighet, arbetsfördelning och ansvar.

Samtidigt som AI-baserad kodgenerering effektiviserar produktionen väcker den grundläggande frågor om kodkvalitet. Kodkvalitet är ett begrepp som är svårt att explicit definiera men enkelt förklarat kan man säga att kodkvalitet kan ses ur två olika perspektiv. För det första måste den genererade koden vara korrekt i förhållande till den specifikation och de instruktioner som uttrycks i prompten. Detta innebär att koden ska implementera avsedd funktionalitet på ett ändamålsenligt sätt, utan att introducera oönskade beteenden eller utelämna funktioner som är nödvändiga för att uppfylla målet. Brister i denna kravuppfyllelse är välkända utmaningar vid AI-baserad kodgenerering. För det andra handlar kodkvalitet om kodens interna egenskaper. Att kod fungerar korrekt är en nödvändig, men inte tillräcklig, förutsättning för hög kvalitet. Koden ska även vara begriplig, underhållbar, effektiv och anpassad till den kontext där den används. Traditionella sätt att utvärdera kodkvalitet bygger i huvudsak på algoritmiska mått såsom komplexitet, storlek och koppling, ofta i kombination med testtäckning och manuell kodgranskning (Sharma och Spinellis, 2020, IV). Dessa metoder har utvecklats inom klassisk mjukvaruteknik och är framtagna för mänskligt skriven kod.

Tidigare forskning genomförd av Sharma och Spinellis (2020, II) visar att flera etablerade kodmått lider av bristande träffsäkerhet och riskerar att ge missvisande resultat. Oliveira et al. (2021, I) argumenterar för att tolkningen av kvantitativa kodmått i praktiken är subjektiv och svår, särskilt i stora kodbaser och vid jämförelser mellan olika mjukvaruversioner.

Den ökande användningen av AI för kodgenerering förstärker denna problematik ytterligare. Studier visar att AI-genererad kod ofta uppvisar goda värden enligt traditionella mått för exempelvis underhållbarhetsindex, cyklomatisk komplexitet och kommentarsfrekvens (Eltabakh, Nabil Soudi och Shawky, 2024, s. 203–205). Samtidigt är det osäkert om dessa mått verkligen fångar kvalitet i en större systemkontext. Vidare saknas det i dagsläget en tydlig konsensus kring hur AI-genererad kod bör utvärderas**,** och om befintliga kodutvärderingsmetoder är tillräckliga eller behöver anpassas för denna nya typ av kodproduktion.

Mot denna bakgrund uppstår frågan hur kvaliteten hos AI-genererad kod bör utvärderas på ett tillförlitligt och meningsfullt sätt: är traditionella algoritmiska mått tillräckliga, eller krävs kompletterande perspektiv som tar hänsyn till specifikation, kontext och mänsklig bedömning?

1.1 Syfte och frågeställningar

Syftet med denna uppsats är att undersöka och analysera hur kvaliteten hos AI-genererad kod kan utvärderas**,** med fokus på att jämföra traditionella algoritmiska kodutvärderingsmått med utvärdering baserad på kravuppfyllelse och specifikation. Arbetet avser att identifiera styrkor och begränsningar hos befintliga kodutvärderingsmetoder samt att undersöka vilka mått och angreppssätt som är mest ändamålsenliga för utvärdering av kod genererad av LLM:er. Studien baseras på en litteraturstudie och en intervjustudie i kombination med en empirisk analys av AI-genererad kod. Detta möjliggör en analys av hur kodkvalitet bör bedömas och utifrån detta kan ett utvärderingsramverk för AI-genererad kod utvecklas. För att uppnå syftet kommer följande frågeställningar undersökas:

- Hur väl fångar traditionella kodutvärderingsmått kvaliteten hos AI-genererad kod?
- Vilka utvärderingsmått är mest relevanta för att bedöma AI-genererad kodkvalitet?
- Vilka anpassningar av mått eller kompletterande metoder behövs för att utvärdera AI-genererad kod på ett tillförlitligt sätt?

2. Uppdragsgivare och kravspecifikation

Detta examensarbete har genomförts i samarbete med Decerno, ett svenskt IT-konsultföretag som utvecklar digitala system och erbjuder konsulttjänster inom IT-området. Företaget arbetar bland annat med skräddarsydda verksamhetssystem och digitala lösningar för kunder inom olika sektorer. Decerno har omkring 150 anställda och är en del av koncernen Addnode Group (Decerno, 2026). Inom ramen för samarbetet har Decerno även fungerat som uppdragsgivare, och examensarbetet utgår därför från ett antal krav som företaget har fastställt. Dessa krav definierar projektets praktiska förutsättningar och dess akademiska kontext.

En grundläggande förutsättning är att den AI-genererade kod som ska utvärderas under detta projekt alltid baseras på och genereras utifrån en välformulerad och detaljerad specifikation. Specifikationen beskriver vad koden ska uppnå ur ett funktionellt perspektiv och är uppdelad i epics, stories och tasks. Ett centralt mål i arbetet är att den genererade kodens funktion ska överensstämma 1:1 med specifikationen, vilket innebär att koden varken får innehålla extra funktionalitet utöver kraven, eller sakna specificerade beteenden. Följaktligen är ett centralt krav att kodutvärderingen fokuserar på att undersöka i vilken utsträckning den genererade koden faktiskt implementerar det som anges i specifikationen. Utöver att koden utvärderas utifrån hur väl den följer en specifikation ska också kvaliteten av den genererade koden bedömas utifrån ett antal algoritmiska mått. Vidare ska utvärderingen vara kompatibel med kod genererad av flera typer av LLM:er. Ett ytterligare krav är att kodutvärderingen ska genomföras på stora, fullständigt genererade kodbaser bestående av backend utvecklad i C# och .NET samt frontend utvecklad i React, TypeScript och JavaScript. Slutligen ställs krav på att själva utvärderingsprocessen ska vara så deterministisk som möjligt. Det innebär att utvärderingsmetoden ska generera identiska resultat vid upprepade körningar. Utifrån dessa krav ska ett automatiserat utvärderingsramverk utvecklas.

3. Tidigare forskning

3.1 Litteraturöversikt

Forskningsfältet som denna studie relaterar till inkluderar flera delområden, däribland kodkvalitet, automatiserad kodgranskning, traditionella kvalitetsmått samt den framväxande forskningen om generativ AI i mjukvaruutveckling. I syfte att skapa en överblick över fältet genomfördes en systematisk litteraturgenomgång för att identifiera etablerade utvärderingsmått och aktuella forskningsdiskussioner. En mer detaljerad beskrivning av urvalskriterier, sökord och databaser presenteras i metodkapitlet. Litteraturöversikten har två separata syften. Dels har den använts för att etablera den teoretiska bakgrunden till arbetet, dels har den fungerat som grund för att identifiera vilka kvalitetsaspekter och utvärderingsmått som är relevanta att inkludera i studiens fortsatta analys och i utformningen av utvärderingsramverket.

3.2 Utvärdering av kodkvalitet

Traditionellt har kodkvalitet utvärderats genom en kombination av algoritmiska mått, metoder för testtäckning samt manuell granskning (Sharma och Spinellis, 2020, IV). Dessa angreppssätt har vuxit fram inom klassisk mjukvaruteknik och har utvecklats utifrån mänskligt skriven kod. Forskningen visar dock att kodkvalitet är ett mångdimensionellt begrepp där ingen enskild metod är tillräcklig på egen hand. I följande kommer information om nuvarande forskningsläget att presenteras.

3.2.1 Algoritmiska mått för kodkvalitet

I detta avsnitt presenteras de algoritmiska mått för kodkvalitet som ligger till grund för den fortsatta analysen i uppsatsen. Dessa mått används inom både forskning och industri för att beskriva olika aspekter av kodkvalitet. Det är viktigt att notera att flera av måtten kan definieras och beräknas på olika sätt beroende på sammanhang och verktyg.

Cyklomatisk komplexitet är ett mått som utvecklades under 1970-talet, som mäter komplexiteten i exekveringsflödet i en kod genom att beräkna antalet olika exekveringsvägar genom en metod eller funktion. Det är därmed också ett minimivärde för antalet tester som behövs för att nå en hundraprocentig testtäckning. Måttet beräknas något olika i olika studier och sammanhang, men ursprungligen är det definierat på antalet noder och kanter i en tänkt flödesgraf för funktionen. Ett högt värde på cyklomatisk komplexitet tenderar att korrelera med komplex och svårläst kod. (Ciceri et al., 2022, kap. 9)

Kommentarsfrekvens representerar antalet kommentarer per antalet rader kod i ett program och indikerar därmed nivån för dokumentation i koden. Ett högre värde tyder på mer väldokumenterad kod, vilket i sin tur kan underlätta underhåll. (Eltabakh, Nabil Soudi och Shawky, 2024, s. 202)

SLOC står för source lines of code och mäter antalet rader kod i ett program, och är därmed ett mått för kodbasens storlek. Det räknar inte med tomma rader eller rader som endast innehåller kommentarer. (Eltabakh, Nabil Soudi och Shawky, 2024, s. 202; Ciceri et al., 2022, kap. 9).

Koppling, eller på engelska coupling, beräknar i vilken grad olika delar av systemet är beroende av varandra. Hög koppling innebär att en förändring i en komponent riskerar att påverka många andra, vilket försämrar underhållbarheten. (Ford et al., 2021, kap. 3)

Sammanhållning, eller på engelska cohesion, beskriver i vilken utsträckning operationerna inom en komponent hör ihop. En hög grad av sammanhållning innebär att komponenten fokuserar på ett tydligt och väldefinierat ansvar, vilket gör den lättare att förstå, underhålla och vidareutveckla. (Ford et al., 2021, kap. 3)

Komponentstorlek, beskriver hur många exekverbara satser en komponent innehåller. Stora komponenter med många ansvarsområden tenderar att vara svårare att modifiera än mindre, mer fokuserade komponenter. (Ford et al., 2021, kap. 3)

Halstead-mått eller på engelska Halstead-metrics, är en uppsättning storleks- och komplexitetsmått som syftar till att tolka ett programs egenskaper utifrån antalet operatorer och operander samt antalet unika operatorer och operander i källkoden. Operatorer är de språkliga symboler som anger operationer i koden, till exempel addition eller if-satser. Operander är de värden och variabler som operationerna utförs på. Halstead-måttet utvecklades på 1970-talet och syftar till att approximera mjukvarans komplexitet och begriplighet. De mått som ingår i Halstead-mått är programlängd, programvokabulär, volym som mäter programmets informationsinnehåll, svårighetsgrad som indikerar hur svår koden är att läsa, ansträngning som indikerar hur begriplig koden är, samt ett uppskattat antal defekter i koden. (Sharma och Spinellis, 2020, I, II)

Underhållbarhetsindex eller på engelska maintainability index, är ett mått som skapades på 1990-talet och beskriver hur lätt en kod är att förändra över tid. Det handlar om möjligheten att lägga till nya eller anpassa befintliga funktioner, ta bort sådant som inte längre behövs samt genomföra interna förändringar som buggfixar, säkerhetsuppdateringar och ramverksuppgraderingar. Ett system med bra underhållbarhet kan anpassas till nya krav med relativt liten arbetsinsats och låg risk för systemfel. Underhållbarhetsindex är inte ett entydigt eller enkelt mått, utan kan beräknas och bedömas på många olika sätt. Ofta används en kombination av olika algoritmiska mått för att få en helhetsbild av hur lätt ett system är att underhålla. Syftet med att mäta underhållbarhet är att kunna bedöma hur väl ett system stödjer långsiktig utveckling och förändring, samt att identifiera strukturella problem som kan leda till ökade kostnader och längre ledtider vid vidareutveckling. Sammanfattningsvis är underhållbarhetsindex ett centralt och vanligt förekommande algoritmiskt mått som beskriver hur väl ett system klarar förändring över tid (Ford et al., 2021, kap. 3). En vanlig formel som Microsoft använder för att beräkna underhållbarhet bygger på Halstead-volym, cyklomatisk komplexitet och SLOC. Formeln ser ut som följande

$$Underhållbarhetsindex = MAX(0, (171 - 5{,}2 * \ln(Halstead\ volym) - 0{,}23 * (Cyklomatisk\ komplexitet) - 16{,}2 * \ln(SLOC)) * 100/171) \quad (1)$$

där värdet ligger mellan noll och hundra. Ju högre värde desto bättre underhållbarhet. Inom ramen för denna uppsats så används denna formel som definition av det algoritmiska måttet underhållbarhetsindex. (Microsoft, 2025a)

Chidamber–Kemerer-måtten (C&K) är en uppsättning kodmått utvecklade för objektorienterade mjukvara som togs fram under 1990-talet. C&K-måtten introducerades för att mäta centrala objektorienterade egenskaper såsom klasskomplexitet, koppling,

sammanhållning och arv, vilka ansågs ha stor betydelse för underhållbarhet och vidareutveckling av mjukvara. Det finns flera olika mått som ingår i C&K-måtten:

- Weighted Methods per Class (WMC), som summerar metodernas cyklomatiska komplexitet i en klass
- Coupling Between Objects (CBO), som mäter hur många andra klasser en klass är beroende av
- Lack of Cohesion in Methods (LCOM), som uppskattar hur väl metoderna i en klass hänger ihop genom gemensam användning av data
- Depth of Inheritance Tree (DIT), som mäter antalet nivåer i arvshierarkin ovanför klassen
- Number of Children (NOC), som mäter antal klasser som direkt ärver från klassen

C&K-måtten beräknas genom statisk analys av källkod, där man identifierar klasser, metoder, beroenden och arvshierarkier, och används främst för att få indikatorer på kodkvalitet och potentiella underhållsproblem i objektorienterade system. (Sharma och Spinellis, 2020, I, II)

Kodduplicering beskriver förekomsten av identisk kod på flera platser i en kodbas, exempelvis när samma metod återkommer i flera filer eller klasser. Ett centralt ideal inom programvaruutveckling är DRY-principen (”Don’t Repeat Yourself”), vilket innebär att varje beräkning eller logisk operation i ett system så långt som möjligt bör ha en enda tydlig representation. När denna princip bryts försvåras underhållbarhet, skalbarhet och felsökning. (Thomas och Hunt, 2020)

3.2.2 Kvalitetsgranskning med testsviter och statiska kodanalysverktyg

År 2023 publicerade Yetiştiren et al. en artikel där författarna jämför kvalitet på kod skriven av tre olika AI-assistenter – GitHub Copilot, Amazon CodeWhisperer och ChatGPT. Chatbottarna får i uppgift att koda lösningar till 164 problem som är formulerade i datasetet HumanEval. I datatestet finns även i snitt 7,7 tester per problem eller koduppgift. Den genererade kodens kvalitet utvärderades sedan utifrån aspekterna validitet, korrekthet, säkerhet, pålitlighet och underhållbarhet. (Yetiştiren et al., 2023, s. 2, 9)

Studien inkluderade att testerna genomfördes på flera kodgenereringar från flera modeller från respektive verktyg, samt med anpassningar i prompten, såsom att inkludera eller exkludera funktionsnamn och beskrivning av vad funktionen ska göra, för att undersöka skillnaden i outputens kvalitet. Resultaten visade att samtliga agenter presterade sämre i de fall där de inte hade tillgång till en funktionsbeskrivning i inputen, vilket belyser vikten av att tillhandahålla en tydlig problemformulering för att agenterna ska kunna generera korrekt kod. Vidare visade studien att både Amazons CodeWhisperer och GitHubs Copilot presterade allt bättre när nyare modeller testades, vilket pekar på potentialen i att använda AI för kodrelaterade uppgifter i framtiden. (Yetiştiren et al., 2023, s. 10, 43)

I studien *Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation* (2024) undersöker Tosi i vilken utsträckning tre olika språkmodeller, ChatGPT-3, -4 och Google Bard (nuvarande Gemini) kunde lösa programmeringsuppgifter i Java hämtade från en universitetstentamen. Utvärderingen av den genererade koden gjordes utifrån två övergripande aspekter. Dels undersöks om

koden är funktionellt korrekt och robust, alltså att den löser det givna problemet. Detta testades genom att författaren satte upp totalt 32 olika testfall, både förväntad input och hörnfall, som undersöktes. För det andra analyseras koden utifrån statiska mått via det statiska kodanalysverktyget SonarQube, där bland annat SLOC, cyklomatisk komplexitet, läsbarhet, underhållbarhetsindex, code smell, dupliceringar, buggar och säkerhetsrelaterade sårbarheter ingick. (Tosi, 2024, s. 1, 6–7)

Resultaten visar att mänsklig inblandning är nödvändig för att styra språkmodellerna mot fungerande lösningar, men pekar samtidigt på att ett nära samspel mellan mänskliga utvecklare och AI-baserade verktyg kan vara ett sätt att höja både kodkvalitet och utvecklingseffektivitet. Därtill lyfter studien behovet av vidare forskning omfattande flera språkmodeller, andra typer av koduppgifter samt mer komplexa och nyanserade utvärderingsmått i syfte att vidareutveckla hur mjukvaruutveckling som genomförs i samverkan mellan människa och AI bör bedömas och utvärderas. (Tosi, 2024, s. 12)

3.2.3 Kontextens betydelse vid kodutvärdering och manuell granskning

Maikantis et al. (2024) menar att en kods estetiska utseende kan få en inverkan på hur dess kvalitet uppfattas av utvecklare, beroende på vad utvecklaren får för första visuella intryck från den. Med anledning av detta har de undersökt sambandet mellan ett kodblocks estetiska utseende och dess kvalitet mätt med traditionella, algoritmiska mått. Resultaten påvisade att estetiska mått såsom enkelhet, symmetri och täthet korrelerar negativt med komplexitet, SLOC och koppling. Det innebär att kod som uppfattas som mer estetisk tenderar att ha mindre cyklomatisk komplexitet och koppling. (Maikantis et al., 2024, s. 1, 9–10)

Flera tidigare studier beskriver att kontexten är avgörande vid utvärdering av kodkvalitet, eftersom olika utvecklingsmiljöer prioriterar olika aspekter. Tonella och Abebe (2009) belyser detta genom att jämföra öppen källkod med kommersiell, sluten mjukvara. De delar upp kodkvalitet i intern kvalitet, det vill säga kvalitet enligt utvecklarna, och extern kvalitet, det vill säga kvalitet för användarna. De menar att det i kommersiella sammanhang finns ett starkt fokus på extern kvalitet, exempelvis genom omfattande system och acceptanstestning för att säkerställa korrekt funktion och frånvaro av krascher, medan intern kvalitet nedprioriteras. I öppen källkod menar de att den interna kvaliteten prioriteras högre eftersom utvecklingen ofta sker av flera utvecklare, ofta geografiskt spridda, och därmed ställer högre krav på kodens begriplighet, struktur och designegenskaper. Tonella och Abebes exempel visar på vad som anses vara god kodkvalitet varierar beroende på kontexten den utvecklas och tillämpas inom. (Tonella och Abebe, 2009, s. 2–3).

3.2.4 AI-stödd kodkvalitetsgranskning

Benjamin Batte beskriver i sin artikel från 2025, att kodutvärdering är kritiskt för att hitta buggar, säkerställa att kodstandarder efterföljs och underlätta samarbete runt kod. Trots det är det en aktivitet som åsidosätts på grund av tids- och resursbegränsningar. Han menar att detta är ett område där automatisering genom AI-verktyg kan avlasta utvecklare, och förklarar att dessa system kan flagga när kodstandarder inte efterföljs och när buggar, syntaxfel och säkerhetsrisker uppstår. Han beskriver att GitHub Copilot och Amazon CodeGuru demonstrerar funktionalitet såsom att generera tester och dokumentation samt ge kod- och refaktoreringsförslag. Han beskriver vidare att AI

generellt kan bidra i övergripande analys då hela system ska utvärderas och kan där upptäcka avvikelser i mönster eller duplicerad logik, vilket bidrar till underhållbarheten. Samtidigt betonar han att AI inte helt kan ersätta mänsklig kodgranskning eftersom den saknar förståelse för kontext, arkitektur- och designval, och därför kan generera funktionellt korrekt kod, som ändå kan misslyckas i den större kontexten i projektet eller applikationen. Han menar vidare att det finns risker med att förlita sig helt på AI eftersom det kan leda till att utvecklare blir bekväma och förlorar det kritiska tänkandet och problemlösningskunskaper. Han menar därför att den optimala kodutvärderingen görs i samarbete mellan människa och AI. (Batte, 2025, Introduction)

Ett alltmer uppmärksammat område inom AI-stödd kodkvalitetsgranskning är *LLM as a judge*, vilket innebär att en språkmodell används för att utvärdera kod som genererats av en annan språkmodell. Istället för att enbart förlita sig på manuella granskningsmetoder eller traditionella metoder med algoritmiska mått, kan en LLM i form av en AI-agent utvärdera koden. Fördelen med LLM-baserade utvärderingsmetoder är att de, enligt författarna, har en stark förmåga att förstå kontexten och de instruktioner och specifikationer som kod har genererats utifrån. Till skillnad från algoritmiska mått behöver LLM-baserade utvärderingsmetoder inte definiera specifika benchmarks på förhand. Detta innebär att man inte måste analysera koden för att förstå vad som är rimliga gränser eller vad som är bra eller dåliga värden för vissa mått för att kunna genomföra en utvärdering. Dessutom kan LLM-baserade utvärderingsmetoder med hjälp av externa verktyg exekvera både frontend- och backendkod för att upptäcka funktionella fel som en LLM annars inte kan upptäcka när den endast läser kod. Därför är metoden särskilt användbar för kodgenereringsuppgifter där facit saknas, där kraven är komplexa eller där mänsklig utvärdering är kostsam och svår att skala (Wang et al., 2025, s. 2427–2429).

År 2024 presenterade Tong och Zhang sitt ramverk CodeJudge för att utvärdera AI-genererad kod. CodeJudge bygger på att en LLM vägleds till att tänka långsamt, vilket innebär att modellen först genomför en stegvis analys av uppgiftens krav och kodens logik och därefter sammanfattar analysen i en bedömning av om koden är korrekt. LLM:n får två promptar i stället för en. Författarna kallar detta för en tvåstegsprompt, vilket innebär att modellen först genomför en analys och därefter, baserat på denna, formulerar sitt svar. Om LLM:n i stället endast får en enda omfattande prompt utnyttjas inte dess resonerande förmåga på samma sätt, utan modellen riskerar att fatta snabba beslut. Detta tillvägagångssätt är inspirerat av hur mänskliga utvecklare ofta genomför kodutvärdering, genom att göra en noggrann analys av uppgiftens instruktioner och därefter utvärdera kodens funktionalitet, för att slutligen landa i ett beslut kring dess korrekthet. (Tong och Zhang, 2024)

Metoden kan även uppskatta graden av korrekthet genom att identifiera typer av fel och väga in deras allvarlighetsgrad. På så sätt kan CodeJudge uppskatta i vilken utsträckning koden överensstämmer med utvecklarens intention och uppgiftsbeskrivningen. Resultaten visar att CodeJudge uppnår högre korrelation med semantisk korrekthet än tidigare metoder, vilket innebär att den fångar hur meningsfull och logiskt korrekt koden är i sitt sammanhang. Studien visar också att ramverket fungerar relativt väl även utan referenskod, vilket gör metoden särskilt intressant i situationer där testfall eller korrekta lösningar saknas att jämföra mot. Samtidigt belyser författarna vissa begränsningar med metoden, däribland att modellen ibland misstolkar kodens logik, förbiser delar av uppgiftskraven eller blir för försiktig i sin bedömning av felhantering. (Tong och Zhang, 2024)

I artikeln *An Agent-based Evaluation Framework for Complex Code Generation* används LLM as a judge specifikt för att utvärdera komplex kod. Med komplex kod avser författarna den kod som uppstår ur realistiska kodgenereringsuppgifter med flera samtidiga krav, vilket resulterar i större kodmängder med många funktioner och potentiella felkällor som kräver stegvis och detaljerad analys. Sådan kod kan inte tillförlitligt bedömas genom en enskild övergripande granskning, utan kräver att varje krav och motsvarande kodsegment analyseras separat (Wang et al., 2025, s. 2428). För att hantera komplex kod introducerar författarna ett agentbaserat utvärderingssystem för komplex kodgenerering. De består av två huvudsakliga steg. Först formuleras en detaljerad utvärderingsplan som bryter ned instruktionerna som den genererade koden utgår från i flera mindre uppgifter samtidigt som kontextuell information för varje uppgift samlas in. I det andra steget låter man flera agenter analysera koden, där de ger ett betyg och en motivering till det satta betyget. Sedan låter man de olika agenterna förhandla och diskutera det individuellt satta betyget, där slutbetyget beräknas som ett medelvärde av de olika betygen (Wang et al., 2025, s. 2429).

3.2.5 Kännetecken för en god kodkvalitetsutvärdering

För att skapa en mer systematisk och enhetlig grund för kvalitetsutvärdering har International Organization for Standardization (ISO) utvecklat ett standardiserat ramverk för kvalitetsutvärdering. Standarden definierar ett antal kvalitetsmått som kan användas för att kvantitativt bedöma kodens egenskaper och är avsedd att stödja såväl kravspecifikation som uppföljning och utvärdering av kvalitet under och efter utvecklingsprocessen. Den riktar sig till aktörer inom mjukvaruutveckling och syftar till att skapa en gemensam terminologi och en enhetlig struktur för kvalitetsbedömning. Ramverket omfattar både interna och externa kvalitetsaspekter där interna mått avser hur koden är skriven utifrån aspekter som till exempel komplexitet och underhållbarhet. Extern kvalitet fokuserar istället på kodens funktionalitet för användaren och dess beteende vid exekvering. Genom att kombinera dessa perspektiv möjliggör standarden en strukturerad och heltäckande utvärdering av mjukvarans kvalitet. (ISO/IEC 25023, Introduction och s. 2–3)

De kvalitetsmått som ingår i ramverket omfattar funktionalitet, prestanda, kompatibilitet, tillförlitlighet, användarvänlighet, säkerhet, underhållbarhet. Funktionalitet avser i vilken utsträckning mjukvaran uppfyller specificerade krav och levererar korrekta och fullständiga funktioner. Detta innefattar att systemet producerar korrekta resultat och att funktionerna är ändamålsenliga i relation till användarens behov. Prestandaeffektivitet mäter hur effektivt systemet använder tillgängliga resurser, exempelvis svarstid, resursanvändning, kapacitet, samt om dessa uppfyller uppställda krav under givna förhållanden. Kompatibilitet avser systemets förmåga att samexistera och samverka med andra system och komponenter utan att orsaka konflikter eller försämrad funktion. Tillförlitlighet mäter systemets stabilitet och förmåga att upprätthålla korrekt funktion över tid, exempelvis genom låg felfrekvens, hög tillgänglighet samt förmåga till återhämtning vid fel eller avbrott. Vidare omfattar ramverket användbarhet, vilket avser hur lätt systemet är att förstå, lära sig och använda för avsedda användare. Detta inkluderar aspekter som tydlig struktur, intuitiv interaktion och stöd för att minimera användarfel. Säkerhet behandlar skyddet av information och data, exempelvis genom åtkomstkontroll, integritetsskydd, konfidentialitet och spårbarhet, och syftar till att minska risken för obehörig åtkomst eller manipulation. Slutligen avser underhållbarhet hur lätt mjukvaran kan analyseras, modifieras, testas och vidareutvecklas över tid. Detta

inkluderar kodens struktur, modularitet, testbarhet och graden av koppling mellan komponenter. (ISO/IEC 25023, s. 32–35)

ISO:s ramverk prioriterar inte någon enskild kvalitetsdimension framför de andra i absoluta termer, utan betonar att samtliga kvalitetsmått är relevanta och att deras relativa betydelse beror på mjukvarusystemets kontext och användningsområde. Genom att strukturera kvalitetsutvärderingen kring dessa definierade områden och tillhandahålla etablerade mått för respektive dimension bidrar ISO:s modell till en systematisk och jämförbar bedömning av vad som kan anses vara god mjukvarukvalitet. (ISO/IEC 25023, Introduction)

Nelson och Schumann (2004) beskriver att listan över alla aspekter av kvalitet som kan utvärderas i en kodutvärdering är lång. Det kan innefatta allt från övergripande dokumentation till syntaxdetaljer i källkoden. De menar att detta medför att ett urval av vilka aspekter en kodutvärdering ska innehålla görs, och hur detta utformas är ofta beroende av utvärderarens individuella erfarenheter och kunskaper. Detta skapar variation för kvalitetskontrollen, vilket i sin tur medför frågor kring dess trovärdighet. (Nelson och Schumann, 2004, s. 2, 9)

Vidare beskriver författarna att för att en kodutvärdering ska vara trovärdig behöver de undersökta aspekterna tillsammans ge en god täckning, både i form av att få ett representativt och tillräckligt stort urval av kodrader som granskas och att urvalet av olika typer av problem och riskområden ska vara brett. Slutligen beskriver de att kodutvärdering fångar omkring hälften av alla defekter, men att ytterligare verifierings- och valideringsmetoder är nödvändiga för att säkerställa en sammanlagd trovärdig kodutvärdering och bra kodkvalitet. Författarna visar att de aspekter som utvecklare har svårast att identifiera vid kodutvärdering är avvikelser relaterade till synkronisering av flertrådad kod, det vill säga kod med flera exekveringsflöden. (Nelson och Schumann, 2004, s. 2, 9)

Börstler et al., har i sin studie *Developers talking about code quality* (2023) intervjuat totalt 34 utvecklare, studenter och lärare med i snitt 5,8 års arbetslivserfarenhet av programmering för att undersöka deras syn på kodkvalitet och hur de uppfattar vad som är bra respektive dålig kod från utvecklares perspektiv. Vidare beskriver Börstler et al. (2023) att det finns forskning som visar på att kvalitetsproblem i koden ökar den kognitiva belastningen hos utvecklaren, vilket i sin tur får en negativ påverkan på deras prestationer. I studien får utvecklare utvärdera kod genom att berätta och resonera kring vilka aspekter de anser är mer eller mindre viktiga för kodkvalitet, snarare än genom att mäta olika algoritmiska index, och ge rekommendationer för hur man uppnår god kodkvalitet. (Börstler et al., 2023, s. 2)

Studiens resultat visar att utvecklare strävar efter begriplighet och underhållbarhet, vilket innebär att kod ska vara välstrukturerad, enkelt läsbar och dokumenterad på ett lämpligt sätt. Samtidigt som kodkommentering beskrivs som viktigt, betonas att det kan bli en distraktion om det görs i för stor utsträckning. Korrekt och genomtänkt namngivning tillsammans med lämplig struktur kan bidra med mer begriplighet. Slutligen beskriver författarna att kodkvalitet är komplext och bör utvärderas på flera nivåer, eftersom god kvalitet på detaljnivå, i form av exempelvis lämplig namngivning och kommentarer, inte nödvändigtvis medför god övergripande kvalitet på en högre, strukturell nivå, och vice versa. (Börstler et al., 2023, s. 21–22)

3.3 Brister med nuvarande kodutvärderingsmetoder

I artikeln *Do We Need Improved Code Quality Metrics?* identifierar Sharma och Spinellis ett flertal grundläggande brister i dagens etablerade kodutvärderingsmetoder. En central kritik är att många vanligt förekommande kodmått uppvisar bristande förankring, det vill säga att de inte mäter det kvalitetsbegrepp de påstår sig representera. Författarna lyfter fram Halstead-mått och cyklomatisk komplexitet som två exempel på mått med bristande träffsäkerhet, vilket leder till att resultaten ofta blir missvisande eller svåra att tolka i praktiken. Cyklomatisk komplexitet reducerar komplexitet till antalet oberoende exekveringsvägar, vilket inte nödvändigtvis speglar hur komplex koden upplevs av människor eller hur svår den är att underhålla. Till exempel presterar cyklomatisk komplexitet speciellt dåligt vid analys av switch-satser, där komplexiteten enligt måttet inte motsvarar den mänskliga förståelsen. En switch-sats har många exekveringsvägar, men kräver låg kognitiv belastning för en människa att förstå. Därför blir den cyklomatiska komplexiteten missvisande. Eftersom underhållbarhetsindex bygger på cyklomatisk komplexitet så anses den också lida av samma brister. (Sharma och Spinellis, 2020, II)

Vidare kritiserar författarna Halstead-måtten för att också sakna en stabil teoretisk grund och för att bygga på ett alltför förenklat synsätt på mjukvara. Halstead-måtten reducerar programkomplexitet till beräkningar av operatorer och operander. Författarna menar att Halstead-måtten är otillräckliga för att fånga de strukturella egenskaper som präglar modern mjukvara. De anses vara för primitiva eftersom de utvecklades under en tid då mjukvarusystem var betydligt mindre och enklare än dagens system. Genom att enbart fokusera på ytliga symbolräkningar misslyckas måtten med att fånga centrala programegenskaper såsom kontrollflöden, datastrukturer samt modul- och arkitekturstruktur, vilket gör måttet begränsat för moderna kodbaser. Kontrollflöden, exempelvis nästlade villkor, loopar och alternativa exekveringsvägar, påverkar i hög grad hur komplex kod är att förstå, men behandlas inte meningsfullt av Halstead-måtten. På motsvarande sätt gör måtten ingen skillnad mellan enkla och komplexa datastrukturer, såsom listor, träd eller hash-tabeller, trots att dessa har stor inverkan på både begriplighet och underhållbarhet (Sharma och Spinellis, 2020, II).

Slutligen menar författarna att de traditionella kodutvärderingsmåtten generellt endast utvärdera koden på metod- och klassnivå, medan de bortser från att utvärdera på arkitekturnivå, alltså hur koden är uppdelad i moduler och komponenter. Detta gör det svårt att bedöma kvaliteten hos komponenter, subsystem och hela mjukvarusystem, trots att dessa nivåer ofta är avgörande för långsiktig underhållbarhet. (Sharma och Spinellis, 2020, II)

Oliveira et al. (2021) kritiserar i sin artikel traditionella mått för utvärdering av kodkvalitet och menar att de kan vara svårtolkade. Trots att flera mått bygger på kvantitativa data så menar de att tolkningen av dessa värden i praktiken är subjektiv och starkt kontextberoende. De framhåller att det ofta inte är trivialt att avgöra huruvida ett mätvärde ska betraktas som högt eller lågt, bra eller dåligt, särskilt vid analys av stora kodbaser – eller vid jämförelser mellan olika mjukvaruversioner. När kodbasen är stor blir det svårt för en utvecklare att manuellt avgöra vad som är ett normalt, avvikande eller problematiskt värde. Desamma gäller för tolkningar av olika versioner där det är svårt att

se om ett ökat eller minskat värde leder till positiva eller negativa förändringar. Ett värde som är högt i en version kan vara normalt i en senare version, särskilt när systemet växer. Därför blir fasta trösklar missvisande. Detta innebär att numeriska mått i sig inte ger tillräckligt stöd för att dra meningsfulla slutsatser om kodkvalitet utan ytterligare tolkningsstöd krävs. Därför menar författarna att denna inneboende subjektivitet i måttbaserad kodanalys skapar en begränsning hos traditionella kodutvärderingsmetoder – och att det därför krävs kompletterande angreppssätt som bättre speglar hur människor uppfattar och tolkar kvalitet (Oliveira et al, 2021, abstract, I).

Börstler et al. (2023) menar, i likhet med Oliveira et al. (2021), att kodkvalitet är ett komplext begrepp som inte kan reduceras till ett entydigt numeriskt mått, utan att det kräver mänsklig tolkning. Genom intervjuer med utvecklare visar de att kvalitet i praktiken bedöms utifrån mänsklig uppfattning och erfarenhet, där egenskaper som struktur, läsbarhet, dokumentation och förståelighet värderas högt, trots att dessa aspekter är svåra att fånga med etablerade kodmått. Exempelvis anses korrekt och genomtänkt namngivning och kodkommentering vara viktigt. Samtidigt betonar författarna att för mycket kodkommentering kan leda till distraktion. Vad som utgör en lagom och ändamålsenlig nivå av kommentering beror därför på kontext och mänsklig bedömning, vilket gör denna aspekt svår att uttrycka med ett enskilt numeriskt värde (Börstler et al., 2023, s. 2, 21–22).

Martin et al. (2026) menar att kodkommentarer hade varit överflödiga om kodspråken vore tillräckligt uttrycksfulla De menar att nödvändiga kommentarer endast är de som skrivs då man misslyckas med att skriva tillräckligt tydlig kod. Därmed hade kodkommentarer alltså inte behövts alls i perfekt kod. Författarna uppmuntrar att man istället för att kommentera kod ska fundera på olika sätt att omformulera eller omstrukturera koden så att det blir mer intuitivt för läsaren. De förklarar också att kodkommentarer ofta är svåra att förstå utan att man förstår den skrivna koden. Kommentarer och kod behöver därför gå hand i hand. (Martin et al., 2026, kap. 5)

Begriplighet är ett mått som författarna Scalabrino et al., har undersökt i artikeln *Automatically Assessing Code Understandability: How Far Are We?* där de har genomfört en empirisk studie av kodförståelse. Författarna analyserar totalt 121 existerande kodutvärderingsmått och låter 63 utvecklare bedöma sin faktiska och upplevda förståelse av 50 Java-kodfragment. (Scalabrino et al., 2021, s. 595)

Resultaten från artikeln visar att i stort sett inga av de undersökta kodutvärderingsmåtten uppvisar någon meningsfull korrelation med upplevd eller faktisk begriplighet. Detta är särskilt intressant då utvecklare spenderar cirka 70 procent av sin arbetstid på att försöka förstå befintlig kod. Alltså finns det, enligt författarna, inget mått som kan mäta den mest centrala faktorn i mjukvaruutveckling. Artikeln visar även att läsbarhet och begriplighet är två skilda begrepp. Kod kan vara lätt att läsa men ändå svår att förstå, exempelvis på grund av komplexa beroenden eller dålig dokumentation. Därmed ifrågasätts antagandet att traditionella komplexitets- och läsbarhetsmått kan användas för att undersöka kods begriplighet (Scalabrino et al., 2021, s. 595). Författarna drar slutsatsen att vi i dagsläget är långt ifrån att kunna mäta mänsklig kodförståelse automatiskt, och att mänsklig kodförståelse är starkt kontextberoende och varierar mellan utvecklare, där olika individer lägger olika vikt vid olika aspekter. De menar att begripligheten av kod beror i högre grad av utvecklaren än på den kod som utvecklaren granskar. Denna subjektivitet är svår att fånga med traditionella algoritmiska mått och därför menar författarna att dessa mått är begränsade. (Scalabrino et al., 2021, s. 605, 610–611)

3.4 Karakteristiska egenskaper hos AI-genererad kod

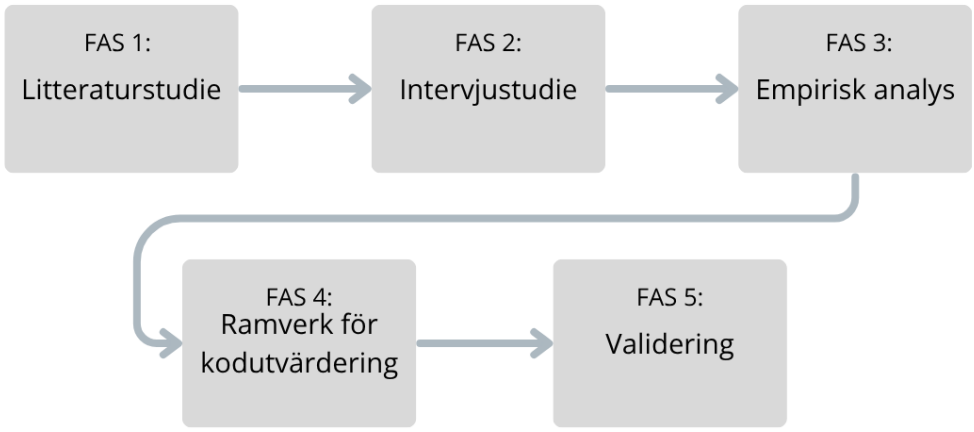
Martin et al. (2026) beskriver att LLM:er idag, på klassnivå, kan producera kod som är omkring 80 procent korrekt, och att man genom mänskligt stöd kan nå bättre resultat. Genom att tillhandahålla exempel och beskrivningar kan man tvinga den till viss kontinuitet exempelvis genom att prompta att varje funktion eller metod endast ska ha en uppgift eller syfte, och undvika bieffekter. De reflekterar kring att det är enligt instruktioner som dessa som många lär sig att koda. Författarna menar att vi har passerat stadiet där AI används som en kodassistent rad för rad, och att det idag går att prompta fram hela moduler genom att tillhandahålla exempel och tester. De betonar att fokus borde ligga på den övergripande systemarkitekturen snarare än på enskilda kodrader. Genom att utforma system på ett modulärt sätt kan hela moduler och funktioner läggas till och bytas ut utan att riskera att påverka eller förstöra befintlig kod. Det bidrar till ett mer flexibelt och underhållbart system, där ny funktionalitet kan utvecklas och integreras som separata delar i helheten. (Martin et al., 2026, kap. 13)

Eltabakh, Nabil Soudi och Shawky undersöker i sin studie *Quality of AI-Generated vs. Human-Generated Code* 2700 koduppgifter i Python med tillhörande mänskligt skrivna lösningar. De lät därefter ChatGPT4 generera lösningar till samma koduppgifter och analyserade sedan samtliga lösningar med traditionella kodutvärderingsmått. Genom att träna en maskininlärningsmodell på både mänskligt skriven och AI-genererad kod kunde författarna dels klassificera om en given kodlösning var skriven av en människa eller av AI, dels undersöka vilka egenskaper som hade störst betydelse för klassificeringen. (Eltabakh, Nabil Soudi och Shawky, 2024, s. 200–201)

Resultaten visade att noggrannheten för klassificeringen var cirka 88 procent. De algoritmiska måtten som var viktigast i klassificeringen var i inbördes ordning kommentarsfrekvensen, underhållbarhetsindex, cyklomatisk komplexitet och antalet parametrar. Med andra ord tyder studiens resultat på att det finns olikheter i mänskligt skriven respektive AI-skriven kod som gör att det går att klassificera hur koden är genererad genom att undersöka dessa algoritmiska mått. Vidare förklarar författarna att trender som upptäcktes var att den AI-genererade koden tenderade att ha fler kommentarer än mänskligt skriven kod. Den mänskligt skrivna koden hade något lägre värde på underhållbarhetsindexet än den AI-genererade koden och detta kan bero på att AI-genererad kod till sin natur är mer strukturerad. Slutligen menade de att den AI-genererade koden hade lägre cyklomatisk komplexitet, vilket tyder på enklare strukturer och ett mer rättframt upplägg, samtidigt som det kan innebära en minskad flexibilitet för att kunna hantera mer komplexa problem och nyanserade uppgifter. (Eltabakh, Nabil Soudi och Shawky, 2024, s. 203)

4. Metod

Studiens metod kan delas upp i fem faser: litteraturstudie, intervjustudie, empirisk analys, framtagande av utvärderingsramverket samt en validering av utvärderingsramverket. Faserna genomfördes sekventiellt, där resultatet från varje fas låg till grund för den efterföljande. En schematisk översikt av studiens faser och deras inbördes relationer presenteras i figur 1. I följande avsnitt beskrivs varje fas mer ingående, tillsammans med de metodologiska överväganden och avgränsningar som gjorts.



Figur 1. Schematisk illustration av studiens faser.

4.1 Litteraturstudie

Den första fasen bygger på en strukturerad litteraturstudie, där artiklar och konferensbidrag har hämtats från databasen IEEE Xplore med sökfrasen "Abstract":"code" AND "Abstract":"quality" AND "Abstract":"evaluate". Sökningen genomfördes vid två tillfällen, dels filtrerad på publikationer från åren 1990–2022, i syfte att exkludera studier publicerade efter att de stora generativa AI-modellerna blivit allmänt tillgängliga, dels filtrerad på publikationer från 2023 och framåt, för att inkludera dessa. I databasen Google Scholar användes sökfrasen “evaluation of the quality of AI generated code”, dels utan tidsbegränsning, dels filtrerad på publikationer från 2025 och framåt, för att omfatta den allra senaste forskningen. Detta möjliggjorde att även nyare studier som ännu inte hunnit publiceras i IEEE kunde beaktas i studien. Utöver detta har ytterligare studier valts ut och inkluderats genom snöbollsurval.

Litteraturstudien har fokuserat på traditionella, algoritmiska kodutvärderingsmått i syfte att etablera en teoretisk grund och identifiera utvärderingskriterier som är allmänt accepterade inom aktuell forskning och praxis. Därtill har nyare forskning som berör generativ AI och AI-stödd mjukvaruutveckling studerats.

4.2 Intervjustudie

Litteraturstudien visade på kodkvalitet som ett starkt kontextberoendebegrepp. Med anledning av detta efterföljdes litteraturstudien av intervjuer med nio utvecklare på Decerno, i syfte att fånga den specifika kontext detta arbete tar plats inom. Därtill genomfördes en gruppintervju med ett team som arbetar i ett helt AI-drivet projekt, i syfte att få en inblick i ett praktiskt exempel där utvecklare behöver hantera kvalitetsfrågor för AI-genererad mjukvara.

De intervjuade deltagarna hade varierande yrkeserfarenhet inom mjukvaruutveckling, med en genomsnittlig arbetserfarenhet om 10,7 år. Valet av intervjuer som datainsamlingsmetod motiverades av möjligheten att ta del av svåråtkomlig information som inte nödvändigtvis är dokumenterad i skriftliga källor (Eriksson och Wiedersheim-Paul, 2011, s. 99–100). Intervjuerna genomfördes i semistrukturerad form, då vissa i förväg definierade frågor krävdes för att möjliggöra jämförelser mellan intervjuerna. Samtidigt gav den semistrukturerade intervjumetoden utrymme att ställa följdfrågor som

kunde uppstå under samtalens gång (Busetto et al., 2020, s. 3; Eriksson och Wiedersheim-Paul, 2011, s. 99–100).

En känd risk vid användning av intervjuer som datakälla är den så kallade intervjueffekten, vilken innebär att respondenter omedvetet kan anpassa sina svar efter vad de uppfattar som socialt önskvärt eller efter vad de tror är det ”korrekta” svaret. Även ledande frågor kan påverka respondenternas svar, vilket i förlängningen kan ha en negativ inverkan på datamaterialets tillförlitlighet (Eriksson och Wiedersheim-Paul, 2011, s. 99–100). För de enskilda intervjuerna formulerades fyra intervjufrågor, se avsnitt 4.2.1. Till gruppintervjun utformades en intervjuguide med övergripande teman och frågor kopplade till studiens forskningsfrågor, vilket är karakteristiskt för semistrukturerade intervjuer (Busetto et al., 2020, s. 3). Intervjuguiden finns bifogad som Bilaga A Intervjuguide. Majoriteten av intervjuerna genomfördes på plats på Decernos kontor i Stockholm och Uppsala, medan ett mindre antal intervjuer genomfördes på distans. Detta kan ha en inverkan på vilken typ av information som framkommer under intervjuerna. Samtidigt kan det vara viktigt att belysa att Decerno är en arbetsplats med stark digital arbetskultur, så huruvida intervjuerna är genomförda på plats eller digitalt bör inte ha lika stor påverkan på resultatet då medarbetarna är vana och trygga med att delta i videomöten.

I denna studie användes utvecklare anställda vid Decerno som intervjuobjekt, eftersom de arbetar med mjukvaruutveckling på daglig basis och därmed kontinuerligt förhåller sig till frågor om kodkvalitet. Den litteratur som behandlas i avsnitt 3 visade att kodkvalitet är ett kontextberoende och ett subjektivt begrepp, där bedömningen av huruvida en specifik kod uppfattas som kvalitativ i hög grad beror på vem som genomför utvärderingen (Scalabrino et al., 2021, s. 610). Genom korta intervjuer möjliggjordes därför en undersökning av vilka aspekter av kodkvalitet som utvecklarna på Decerno ansåg vara särskilt viktiga, samt hur begreppet god kodkvalitet definierades och tillämpades i praktiken.

En central anledning till att kombinera intervjustudien med en litteraturstudie var den begränsade tillgången på liknande empiriska intervjustudier, särskilt sådana genomförda i närtid. Detta bedömdes som särskilt relevant då AI är ett forsknings- och tillämpningsområde i snabb utveckling, där både modeller och verktyg förändras i hög takt. Nya AI-baserade utvecklingsverktyg lanseras frekvent, vilket medför att etablerade arbetssätt och praxis inom mjukvaruutveckling kontinuerligt omprövas. Det som tidigare betraktades som vedertaget behövde därmed inte nödvändigtvis vara giltigt vid en senare tidpunkt. Mot denna bakgrund ansågs det vara av stor vikt att arbeta med aktuellt och färskt empiriskt material, för att studiens resultat skulle vara så relevanta, precisa och användbara som möjligt.

4.2.1 Intervjufrågor till enskilda intervjuer

- Hur lång arbetslivserfarenhet har du av mjukvaruutveckling?
- Vad är bra kodkvalitet för dig och varför? Finns det vissa aspekter du tycker är viktigare än andra?
- Om du svarade effektiv, underhållbar, förståelig, snabb, billig, funktionell, ändamålsenlig, etc. – vad betyder det?
- Hur jobbar du själv med att utvärdera kod vid pull requests?

4.3 Utvärderingsramverkets kontext och grundförutsättningar

Vårt utvärderingsramverk är avsett att tillämpas i en bred, generell kontext och har därför inte utformats med utgångspunkt i ett specifikt projekt. De kodbaser som omfattas av analysen utgörs av större fullstackapplikationer med en övergripande likartad arkitektur, bestående av en backend utvecklad i C# och .NET samt en frontend utvecklad i React, TypeScript och JavaScript. Gemensamt för alla kodbaserna som analyseras är att de är helt AI-genererade och stora, även om den exakta storleken kan variera mellan projekten. Ramverket behöver därför vara tillräckligt generellt för att kunna tillämpas konsekvent mellan flera kodbaser med liknande struktur och teknisk uppbyggnad, utan att vara beroende av projektspecifika implementationer eller strukturer.

Samtidigt innebär kodbasernas storlek och sammansättning att det inte genomgående är metodologiskt lämpligt att inkludera samtliga filer i analysen. För att möjliggöra en meningsfull bedömning av kodkvalitet krävs därför ett urval av vilka delar av kodbasen som inkluderas i analysen. Vissa typer av filer behöver exkluderas, exempelvis filer som inte representerar relevant programlogik. Avgränsningen syftar därmed till att säkerställa att ramverket fokuserar på de delar av systemen som är mest analytiskt relevanta.

4.3.1 Tree-sitter

För att möjliggöra en strukturell analys av källkoden används parserbiblioteket Tree-sitter. Istället för att behandla programkod som ren text omvandlas den till en syntaktisk representation i form av ett abstrakt syntaxträd. I denna representation delas koden upp i språkliga byggstenar, såsom klassdeklarationer, metoder och variabler, där varje del motsvaras av en identifierbar enhet i en hierarkisk struktur. Även språkliga egenskaper såsom åtkomstnivåer, exempelvis public, private, protected och static, representeras explicit i denna struktur och kan därför identifieras (Tree-sitter, u.å). Detta tillvägagångssätt gör det möjligt att analysera kodens uppbyggnad baserat på dess struktur snarare än på textbaserade mönster. Genom att arbeta med en strukturerad modell av programmet kan klassmedlemmar identifieras och särskiljas på ett effektivt sätt. Varje kodfil parsas med den språkspecifika grammatik som motsvarar filens typ, och det resulterande syntaxträdet analyseras.

4.4 Empirisk analys och utvärderingsramverkets utformning

Intervjustudien följdes av en empirisk undersökning, där AI-genererade kodbaser analyserades och utvärderades med de mått som identifierats som relevanta utifrån litteraturstudien och som fångar de kvalitetsaspekter som framkommit i intervjustudien. Undersökningen baserades på två kodbaser, dels en AI-genererad kodbas framtagen med utgångspunkt i ett open source-projekt, dels en AI-genererad kodbas från ett kundprojekt.

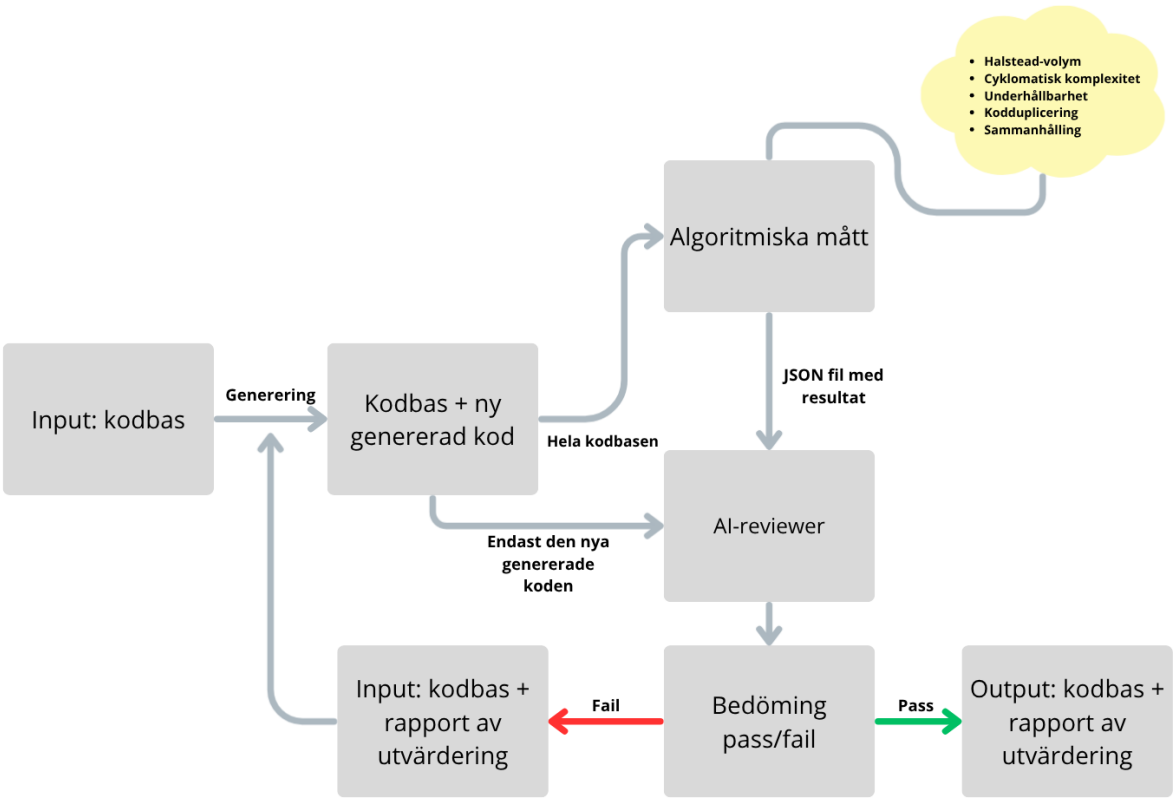
Den open source-baserade kodbas som användes i den empiriska undersökningen utgick från projektet RealWorld. RealWorld används för att skapa bloggplattformar och kännetecknas av att det finns flera olika implementationer som alla följer samma API-specifikation. Detta gör det möjligt att kombinera olika frontend- och backendlösningar på ett enhetligt sätt, vilket innebär att samma fullstackapplikation kan implementeras med olika frontend- och backendspråk (Simons, 2017). Den implementation som användes i den empiriska undersökningen var dock inte densamma som den som senare användes i valideringen i avsnitt 4.9. Även om både den empiriska undersökningen och valideringen

tog sin utgångspunkt i RealWorld, baserades de således på olika implementationer och därmed på olika kodbaser.

Den företagsinterna kodbasen utgjordes av den kod som genererats inom AI first-projektet, vilket beskrivs närmare i avsnitt 5.2. Resultaten av den empiriska undersökningen visade vilka algoritmiska kodkvalitetsmått som genererade mest insikt och bäst fångade kodkvalitet hos AI-genererad kod.

Utifrån resultaten från den empiriska undersökningen fastställdes det slutliga utvärderingsramverket för studien. Utvärderingsramverket omfattar de algoritmiska kodkvalitetsmått som, utifrån resultaten från litteratur- och intervjustudien, bedömdes ge störst analytiskt värde vid utvärdering av AI-genererad kod, både i relation till kodens interna struktur och dess funktion i en större systemkontext. De mått som inkluderades i det slutliga ramverket var sammanhållningsmåtten TCC och LCOM1, kodduplicering, cyklomatisk komplexitet för både enskilda metoder och för hela filer samt Microsofts underhållbarhetsindex enligt ekvation (1) som bygger på Halstead-volym, SLOC och cyklomatisk komplexitet på filnivå.

Dessa mått används som indata till den AI-reviewer som tillämpas som ett kompletterande steg i utvärderingen och som beskrivs närmare i avsnitt 4.8. På så sätt integreras de kvantitativa kodmåtten i en vidare bedömning, där deras utfall kan tolkas i relation till kodens syfte, struktur och kravuppfyllelse. Se figur 2 för en illustration över flödet i utvärderingsramverket.



Figur 2. Illustration av utvärderingsramverkets flöde.

4.5 Sammanhållning för C#-kod

Inom ramen för denna analys kommer begreppet sammanhållning syfta till kopplingen mellan metoder och attribut i C#-kod. Metoder är funktioner som definieras inom en klass och beskriver objektets beteende medan attribut representerar det tillstånd som ett objekt har (Microsoft, 2025b). Analysen av sammanhållningen avgränsas till att endast klasser som innehåller minst två metoder och minst ett attribut analyseras. Sammanhållningsmått bygger på relationen mellan flera metoder och deras gemensamma användning av attribut. Om en klass innehåller färre än två metoder kan inga metodpar bildas. Om den saknar attribut saknas grund för att mäta delat tillstånd. I dessa fall skulle beräkningen sakna analytiskt värde och därför exkluderas sådana klasser från studien.

4.5.1 Definition av metoder

För att möjliggöra en konsekvent analys av sammanhållning behöver det tydligt fastställas vilka språkelement i C# som i denna uppsats klassificeras som metoder. Eftersom olika språkfunktioner kan representera beteende på olika sätt definieras här vilka element som inkluderas i metodbegreppet inom ramen för denna analys. Följande räknas som metoder:

- Instansmetoder
- Överlagrade metoder
- Konstruktorer
- Properties med explicit logik
- Indexerare

Instansmetoder som är deklarerade med åtkomstnivån abstract är exkluderade från analysen eftersom de saknar implementation i den aktuella klassen, och därmed inte innehåller exekverbar kod som kan relateras till klassens interna tillstånd (Microsoft, 2021). Metoder deklarerade som static exkluderas också från sammanhållningsanalysen eftersom de tillhör klassen som helhet och inte är kopplade till en specifik objektinstans (Microsoft, 2025c). Detta görs eftersom sammanhållning i denna studie definieras utifrån relationen mellan metoder och attribut. Instansmetoder med övriga åtkomstnivåer inkluderas i analysen.

Överlagrade metoder, på engelska overloaded methods, är metoder med samma namn men med olika parameterlistor. Detta inträffar när en klass definierar flera versioner av en metod som utför liknande funktioner men tar olika typer eller antal parametrar som indata (Microsoft, 2023a). Trots att metoderna har samma namn betraktas de som skilda implementationer och därför inkluderas de i analysen.

Konstruktorer inkluderas eftersom de innehåller exekverbar kod. En konstruktor är den funktion som anropas vid skapandet av en objektinstans och vars primära syfte är att tilldela initiala värden till objektets attribut samt säkerställa att objektet befinner sig i ett giltigt tillstånd (Microsoft, 2025b).

Properties med explicit logik inkluderas när deras *get*- och/eller *set*-del innehåller exekverbar kod eftersom den då inte enbart representerar lagrat tillstånd, utan även uttrycker beteende genom att utföra beräkningar. En property i C# är en klassmedlem som används för att läsa eller ändra ett objekts data på ett kontrollerat sätt. (Microsoft, 2025d)

Indexerare inkluderas eftersom de innehåller exekverbar kod för att hämta eller tilldela värden och därmed bidrar till klassens beteendestruktur. En indexerare gör det möjligt att behandla ett objekt som om det vore en samling, till exempel en lista eller en array. (Microsoft, 2025e)

4.5.2 Definition av attribut

För att analysen av sammanhållning ska vara konsekvent behöver det även fastställas vilka konstruktioner i C# som betraktas som attribut. Följande inkluderas som attribut inom ramen för denna uppsats:

- Instansvariabler
- Auto-implementerade properties
- Ärvda attribut

Instansvariabler är variabler som deklareras direkt i klassen och som är knutna till en specifik objektinstans. Statiska variabler exkluderas från analysen eftersom de tillhör klassen som helhet och delas mellan samtliga instanser (Microsoft, 2023b). Övriga variabler inkluderas i analysen.

Auto-implementerade properties inkluderas som attribut när de saknar exekverbar kod i sina *get*- och *set*-delar. I dessa fall representerar propertyn ett lagrat tillstånd och betraktas därför som attribut snarare än metoder, då de inte innehåller egen exekverbar kod utan enbart fungerar som en strukturerad representation av data (Microsoft, 2025d).

Ärvda attribut syftar på attribut som deklareras i en basklass och som är åtkomliga i den analyserade klassen. Eftersom en subklass kan använda och manipulera dessa attribut genom sina metoder utgör de en del av objektets totala tillstånd (Microsoft, 2022). Därför inkluderas även sådana attribut i analysen.

4.5.3 Beräkning av sammanhållning

De två mått för sammanhållning som används i denna studie är LCOM1 (Lack of Cohesion of Methods 1) och TCC (Tight Class Cohesion). Båda måtten är etablerade inom objektorienterad kodanalys och bygger på relationen mellan metodpar och deras gemensamma användning av attribut. För varje metod i en klass identifieras vilka attribut metoden använder. Därefter jämförs varje metodpar för att avgöra om de delar minst ett gemensamt attribut. LCOM1 definieras som

$$LCOM1 = \max(0, P - Q), \tag{2}$$

där P är antalet metodpar som inte delar några attribut, och Q är antalet metodpar som delar minst ett attribut. Om P − Q är negativt sätts värdet till noll (Chidamber och Kemerer, 1994, s. 488). Ett lågt LCOM1-värde indikerar hög sammanhållning, eftersom det innebär att många metoder använder gemensamt tillstånd. Ett högt värde indikerar låg sammanhållning och att metoder i större utsträckning arbetar oberoende av varandra. TCC definieras som

$$TCC = Q/(P + Q), \tag{3}$$

där Q är antalet metodpar som delar minst ett attribut och (P + Q) är det totala antalet metodpar i klassen. TCC-värdet ligger mellan 0 och 1. Ett värde nära 1 indikerar hög sammanhållning, medan ett värde nära 0 indikerar låg sammanhållning. (Bieman och Kang, 1995, s. 261)

4.6 Kodduplicering

I den empiriska analysen framkom det att AI tenderar att skriva identiskt duplicerade metoder, vilket är något som skiljer sig mot mänskligt skriven kod. Därför implementerades en koddupliceringsanalys i utvärderingsramverket. Analysen av kodduplicering genomförs på metodnivå. Det innebär att enskilda metoder jämförs med varandra över hela projektets kodbas för att identifiera fall där två eller fler metoder har identisk implementering. För att analysen ska vara meningsfull och undvika falska positiva resultat har ett antal regler satts upp för vilka metoder som inkluderas respektive exkluderas i analysen.

För att en metod ska inkluderas i analysen måste den uppfylla fyra villkor. För det första måste metoden ha en metodkropp. Metoder som saknar kropp, exempelvis abstrakta metoder eller gränssnittsdeklarationer, exkluderas eftersom de inte uttrycker någon logik som kan jämföras. För det andra måste metodkroppen innehålla minst fyra rader exekverbar kod, där tomma rader inte räknas. Denna tröskel används för att exkludera mycket små metoder, eftersom duplicering i sådana fall ofta har begränsad betydelse ur ett underhållbarhetsperspektiv. Det kan antas att det sällan ger några designmässiga vinster i att bryta ut mycket korta funktioner. Tröskeln används därför för att fokusera analysen på duplicering som i större utsträckning kan antas vara relevant för refaktorering. Gränsen sattes till fyra rader kod efter inledande försök med olika gränsnivåer. Att använda tröskeln på fyra rader kod bedömdes ge en rimlig balans mellan alltför många och alltför få träffar. Samtidigt är detta en avvägning som påverkar resultatet, och en annan tröskel hade kunnat vara lika försvarbar beroende på studiens syfte och material.

Utöver storlekskravet exkluderas anonyma funktioner i JavaScript och TypeScript, det vill säga funktioner som saknar ett explicit namn. Anonyma funktioner förekommer frekvent som callback-funktioner, och deras strukturella likhet beror ofta på att de följer vanliga etablerade mönster.

4.6.1 Beräkning av kodduplicering

För varje identifierad metod extraheras metodens kropp, det vill säga den del av metoden som innehåller den exekverbara koden. Alla överflödiga blankrader och radbrytningar skalas bort från metodkroppen, så att skillnader i formateringen inte påverkar jämförelsen. Den normaliserade texten omvandlas sedan med hjälp av MD5-algoritmen till ett hashvärde, vilket är en kort representation av metodens innehåll (Rivest, 1992, s. 1). Två metoder betraktas som duplicerade om deras hashvärden och deras metodnamn är identiska. Kravet på identiskt metodnamn säkerställer att endast metoder med samma avsedda syfte jämförs, eftersom metoder med olika namn typiskt sett representerar olika funktionella ansvarsområden även om deras implementeringar råkar vara strukturellt likvärdiga.

För varje metod som identifieras som duplicerad samlas fem datapunkter in. Metodens namn registreras för att identifiera vilken funktionalitet som har duplicerats. De filer där metoden förekommer dokumenteras för att synliggöra var i kodbasen dupliceringen finns, vilket är viktigt för att kunna bedöma dess spridning och om refaktorering är nödvändigt. Även metodens åtkomstnivå registreras, eftersom denna ger information om metodens synlighet och användningskontext, vilket i sin tur kan påverka bedömningen av hur dupliceringen bör hanteras. Därutöver registreras metodens längd i antal rader samt i hur många filer den förekommer. Detta är relevant eftersom längre duplicerade metoder i regel utgör ett större underhållbarhetsproblem än kortare, samtidigt som en metod som återkommer i många filer indikerar en mer utbredd duplicering. Dessa metoder utgör därmed potentiellt ett större behov av att extrahera den gemensamma logiken till en delad komponent.

4.7 Cyklomatisk komplexitet

Analysen av cyklomatisk komplexitet i denna studie genomförs på filnivå, men med hänsyn till att komplexitet kan vara relevant både som en övergripande egenskap hos en hel fil och som en egenskap hos enskilda metoder. Syftet är därför inte enbart att identifiera filer som i stort är komplexa, utan även att synliggöra specifika metoder med hög komplexitet. Cyklomatisk komplexitet bör tolkas i relation till kodens storlek. En lång fil är inte nödvändigtvis problematisk i sig, förutsatt att dess kod är uppdelad i begripliga och relativt fokuserade metoder. På motsvarande sätt kan även en mindre fil vara problematisk om den innehåller en eller flera mycket komplexa metoder. Det som i första hand betraktas som kritiskt är därför kombinationen av storlek och komplexitet, snarare än något av dessa mått isolerat. (Rosenberg et al.,1998).

I analysen beräknas därför två kompletterande mått: den totala cyklomatiska komplexiteten, som används vid beräkning av underhållbarhetsindex, respektive den cyklomatiska komplexiteten för varje enskild metod. Det första måttet används för att fånga hur komplex en fil är i sin helhet och därmed ge en indikation på den samlade mängden beslutlogik i filen. Detta är relevant i relation till filens storlek, eftersom stora filer med hög total komplexitet typiskt sett är svårare att underhålla och mindre tillförlitliga än mindre filer med lägre cyklomatisk komplexitet. Samtidigt kan små filer med hög komplexitet innebära att koden blir mycket kompakt och därför svår att underhålla och förändra. (Rosenberg et al.,1998)

Det andra måttet används för att fånga komplexiteten hos en enskild metod. Detta mått analyserar varje metod i filen för sig och sparar den cyklomatiska komplexiteten för varje metod. Det möjliggör också att enskilda metoder som är oproportionerligt komplexa kan identifieras, även om filen som helhet inte är särskilt komplex. Kombinationen av dessa två mått gör att man kan skilja mellan filer som är komplexa på grund av att filen som sådan är stor och filer där komplexiteten kan kopplas till ett fåtal komplexa metoder.

För att möjliggöra en användbar bedömning när komplexiteten hos en metod bör betraktas som anmärkningsvärd används en svag tröskel på cyklomatisk komplexitet lika med 25 för att flagga potentiellt problematiska fall. Vid dessa fall instrueras AI-reviewern att den cyklomatiska komplexiteten är väldigt hög och att det är en stark indikation för att metoden bör refaktoreras. Om AI-reviewern ändå rekommenderar att en så pass komplex metod ska behållas måste det ges en tydlig förklaring till varför komplexiteten är oundviklig och varför refaktorering skulle vara skadligt. Att tröskelvärdet satts till just 25

beror på att Microsoft rekommenderar att så komplexa metoder bör brytas upp i flera metoder (Microsoft, 2023c). Syftet är inte att fastslå en absolut gräns mellan komplex och icke-komplex kod, utan att det ska fungera som ett verktyg för snabbt kunna identifiera filer och metoder som bör granskas närmare. Metoder med cyklomatisk komplexitet under 25 kan fortfarande betraktas som tillräckligt komplexa för att AI-reviewern ska rekommendera att metoden refaktoreras. Analysen utgår därmed från principen att logik i möjligaste mån bör fördelas över tydligt avgränsade och begripliga metoder, snarare än samlas i långa och djupt nästlade strukturer.

4.8 AI-reviewer

Som ett komplement till de algoritmiska kodmåtten används i denna studie en AI-reviewer för att bedöma den AI-genererade koden utifrån den övergripande kvaliteten samt kravuppfyllelse i relation till den givna kravspecifikationen. Användningen av en LLM för detta ändamål går i linje med Wang et al.s (2025) resonemang kring LLM as a judge. Författarna lyfter fram språkmodellers förmåga att fånga kontext och tolka de instruktioner och specifikationer som koden genererats utifrån, något som är svårt att uppnå med enbart algoritmiska mått.

AI-reviewern är utformad för att vara modelloberoende och kan konfigureras med olika LLM:er. Valet av LLM görs på förhand inför varje körning, vilket gör det möjligt att anpassa verktyget efter behov och tillgång. Resultaten från de algoritmiska kodmåtten används som en input i AI-reviewern vilket kan antas möjliggör en bredare bedömning av kodkvalitet än vad enbart traditionella mått kan ge. Gränsvärdena tillämpas inte som absoluta värden, utan tolkas i förhållande till den enskilda filens uppgift och funktionalitet. Vad som utgör ett högt eller lågt värde varierar mellan olika sammanhang, och en strikt tröskelbaserad bedömning riskerar därmed att missa nyanser som är beroende av kodens kontext. Genom att låta AI-reviewern göra denna kontextuella tolkning kompletteras de algoritmiska måtten med en mer djupgående analys, där måttens innebörd kan vägas mot kodens faktiska syfte i systemet.

4.8.1 Antagonistisk granskning och kontextuell bedömning

AI-reviewern är utformad som en antagonistisk granskare, vilket innebär att den aktivt söker efter brister och avvikelser snarare än att bekräfta att koden är korrekt. Den får tillgång till den kravspecifikation som den genererade koden ska uppfylla och instrueras att identifiera varje avvikelse mellan krav och implementation. När en avvikelse identifieras citeras det aktuella kravet ordagrant, den specifika kodraden som bryter mot kravet anges, och modellen förklarar vad som förväntades enligt kravet och vad som faktiskt implementerades. Utöver kravuppfyllelse granskar AI-reviewern även om koden innehåller funktionalitet som inte efterfrågades i kravspecifikationen, så kallad over-engineering, och föreslår i sådana fall enklare alternativ.

För de algoritmiska kodmåtten är AI-reviewerns uppgift inte att mekaniskt tillämpa tröskelvärden, utan att avgöra om ett identifierat problem faktiskt är skadligt i sitt sammanhang. Vad gäller kodduplicering kan den algoritmiska analysen identifiera metoder med identisk implementation, men det finns typer av duplicering som är svåra att bedöma algoritmiskt. Exempelvis kan boilerplate-kod eller metoder som innehåller privat logik ha identiska implementationer utan att det är motiverat att bryta ut dem till en gemensam metod. Att bryta ut dem skulle istället skapa onödiga beroenden mellan

orelaterade delar av systemet. AI-reviewern får därför outputen från den algoritmiska dupliceringsanalysen och avgör utifrån instruktioner om en metod faktiskt bör brytas ut, med hänsyn till metodens storlek, hur många filer den förekommer i och vilket syfte den fyller i respektive kontext.

Samma kontextkänsliga bedömning tillämpas för sammanhållning och underhållbarhetsindex. Ett lågt TCC-värde behöver inte innebära att en klass har dålig struktur, eftersom vissa klasser av designmässiga skäl hanterar flera olika och orelaterade uppgifter. En klass som ansvarar för att ta emot en HTTP-förfrågan och vidarebefordra den till rätt tjänst behöver exempelvis hantera autentisering, validering och routing, trots att dessa operationer inte nödvändigtvis delar några gemensamma attribut. Sådana klasser får naturligt låga sammanhållningsvärden utan att det utgör ett kvalitetsproblem.

På motsvarande sätt innebär lågt underhållbarhetsindex eller hög cyklomatisk komplexitet inte automatiskt att koden behöver refaktoreras. AI-reviewern instrueras därför att alltid läsa den faktiska koden innan ett beslut fattas, och att enbart flagga problem när avvikelsen inte är motiverad utifrån kodens sammanhang.

När en generering godkänns av AI-reviewern skapas automatiskt en pull request-rapport som sammanfattar den nya koden, vilka krav som uppfyllts och värdena för de algoritmiska måtten. Utvecklaren kan välja att en kvalitetsloop ska aktiveras om koden underkänns. Det innebär att en ny kodgenerering initieras automatiskt. Då utgår modellen inte enbart från den ursprungliga kravspecifikationen, utan även från den granskningsrapport som genererades vid underkännandet. Rapporten innehåller de specifika anledningarna till att koden underkändes, vilket ger modellen möjlighet att adressera de identifierade bristerna i nästa iteration. På så sätt kan koden successivt förbättras genom upprepade genereringscykler tills ett godkänt omdöme uppnås.

4.8.2 Jämförelsevärden

För att möjliggöra denna bredare bedömning av kodkvalitet som beskrivits ovan ges AI-reviewern algoritmiskt beräknade kvalitetsmått för både den nya genererade koden och för hela den befintliga kodbasen. Vid varje kodgenerering beräknas genomsnittsvärdet för de algoritmiska måtten i den befintliga kodbasen, utan att räkna in den nya koden. Detta görs för att skapa jämförelsevärden som representerar den befintliga kodbasen. Syftet med detta är för det första att möjliggöra en bedömning av hur den nya kodgenereringen påverkar kodbasens kvalitet över tid. Genom att jämföra de nya genereringarnas kvalitet utifrån de algoritmiska måtten med kodbasens genomsnitt, går det att avgöra om den tillagda koden i huvudsak ligger i linje med, förbättrar eller försämrar den befintliga kvalitetsnivån.

För det andra används jämförelsevärdena för att identifiera avvikande kvalitet hos den nya genererade koden som förtjänar en mer noggrann granskning. Om ny kod avviker markant från kodbasens genomsnittsvärden kan detta signalera att filen är särskilt komplex, svårunderhållen eller svagt sammanhållen, även om avvikelsen i sig inte automatiskt innebär låg kvalitet. Det kan finnas legitima skäl för de låga värdena men avvikelsen motiverar en mer noggrann granskning av den faktiska koden. Genomsnittet fungerar alltså som ett filter för att identifiera vad som förtjänar uppmärksamhet, inte som ett kriterium som ensamt avgör bedömningen.

AI-reviewern får också tillgång till varje enskild fils värden för de algoritmiska måtten som finns i den existerande kodbasen. Detta skapar ytterligare kontext, eftersom modellen då kan jämföra ny genererad kod med befintliga kod som har liknande ansvar, struktur eller funktion i systemet. Vilket gör det möjligt att bedöma vad som är ett rimligt kvalitetsvärde för just den enskilda filen i just den aktuella kodbasen, snarare än att utgå från generella tröskelvärden.

En metodologisk begränsning med detta tillvägagångssätt är att kontexten i sin helhet hämtas från den befintliga kodbasen. När ett helt nytt projekt genereras i ett tomt repo saknas således kontextuell förankring, och jämförelsevärdena blir i förlängningen själva AI-genererade. Att komplettera kontextkällan med projektspecifika filer eller explicit definierade arkitekturregler skulle kunna utgöra ett sätt att stärka kontextens tillförlitlighet, särskilt i tidiga projektskeden

4.8.3 Tvåstegs-prompt

För att stärka tillförlitligheten i AI-reviewern används en tvåstegsmetod för prompting, som går i linje med vad Tong och Zhang beskriver som långsamt tänkande. En sådan uppdelning är viktig eftersom vissa språkmodeller tenderar att fatta snabbare och mindre tillförlitliga beslut när analys och bedömning sker i samma prompt. En uppdelad så kallad tvåstegs-prompt tvingar i högre grad att först resonera systematiskt och därefter fatta beslut (Tong och Zhang, 2024). I det första steget får AI-reviewern en input med kravspecifikation för den genererade koden, den nya genererade koden, resultaten för de algoritmiska kodmåtten för hela kodbasen, jämförelsevärden från den befintliga kodbasen, samt instruktioner för hur varje mätvärde ska tolkas. Utifrån detta genomförs en analys som sparas i en strukturerad JSON-fil med information om identifierade problem, styrkor och kvalitetsaspekter, utan att något slutgiltigt beslut fattas.

I det andra steget skickas denna analys i form av en JSON-fil vidare till ett nytt modellanrop, där modellen enbart får i uppgift att utifrån den redan genomförda granskningen och fördefinierade beslutsregler avgöra om resultatet ska bedömas som godkänt eller underkänt. Dessa beslutsregler är delvis hårda och delvis kontextkänsliga. För kodduplicering och cyklomatisk komplexitet per metod används absoluta regler, där ett enda fynd som indikerar behov av extrahering respektive refaktorering automatiskt leder till underkänt omdöme. Huruvida ett fynd ska leda till extrahering eller refaktorering avgörs av AI-reviewern i det första steget utifrån kontexten och om extrahering eller refaktorisering faktiskt vore skadlig eller onödigt.

Det övergripande omdömet i det andra steget bygger däremot på en mjukare bedömning, där koden godkänns så länge den uppfyller kärnkraven och inte uppvisar kritiska brister såsom uteblivna krav, buggar som leder till körningsfel, säkerhetssårbarheter, arkitekturproblem eller allvarliga kvalitetsproblem. För dåliga värden på måtten sammanhållning och underhållbarhetsindex tillämpas dessutom en kontextkänslig regel, där modellen måste väga in om ett identifierat problem faktiskt är skadligt eller om det är motiverat utifrån kodens sammanhang. Sammantaget innebär detta att utvärdering kräver en mer nyanserad och kontextberoende bedömning, med en medveten försiktighet att hellre fria än fälla vid osäkerhet.

4.9 Validering

För att undersöka i vilken utsträckning det framtagna utvärderingsramverket fångar kodkvalitet för kontexten det är utformat för genomfördes en ansats till validering. Tre utvecklare med olika lång arbetslivserfarenhet valdes, varav en junior och en senior. Detta urval gjordes för att fånga de olika perspektiv på kodgranskning som kan finnas beroende av utvecklarens arbetslivserfarenhet. Det bör samtidigt noteras att antalet utvecklare i valideringen är begränsat. Ett större urval hade potentiellt kunnat ge en mer robust validering, men inom ramen för denna uppsats bedömdes tre utvecklare som ett rimligt antal.

Alla tre utvecklare fick genomföra samma kodgranskning av fyra olika AI-genererade kodsegment. Två av dessa var så kallade dummy tasks, genererade i syfte att efterlikna extremfall där kvalitetsbrister medvetet implementerats i koden. Detta gjordes i syfte för att undersöka om, och i så fall hur ramverket respektive utvecklarna upptäckte och rekommenderade vidare åtgärder för att hantera dessa. De övriga två var genererade kodsegment från verkliga projekt. Som underliggande LLM för AI-reviewern vid valideringen valdes MiniMax m2.7. Eftersom AI-reviewern är utformad för att vara modelloberoende var detta val inte styrt av tekniska begränsningar i ramverket, utan grundades främst på praktiska överväganden kring kostnad. MiniMax m2.7 är väsentligt mer kostnadseffektiv än mer avancerade alternativ såsom Claudes Opus-modeller (Artifical Analysis, 2026). En begränsning med detta val är att MiniMax m2.7 inte tillhör de mest kraftfulla modellerna på marknaden, vilket innebär att valideringens resultat kan ha påverkats av modellens kapacitet. Vid användning av en kraftfullare LLM hade AI-reviewern potentiellt kunnat fånga fler kvalitetsaspekter. Detta är något som bör beaktas vid tolkning av valideringens resultat.

Det första projektet som användes vid valideringen var open source-projekt RealWorld. Det andra projektet som användes vid valideringen är en AI-genererad version av ett internt administrativt system på Decerno som heter Maya. Det används för att hantera och administrera medarbetares tid- och frånvarorapportering. Originalsystemet är helt internutvecklat av Decerno och byggdes 2001.

Det tredje projektet som användes vid valideringen kallas ShipmentRouter och det är utifrån detta projekt som de så kallade dummy tasksen har genererats. Koden i ShipmentRouter är AI genererad men instruktionerna för genereringen är manuellt skrivna av författarna. Genom att författarna själva styrde över instruktionerna för den kod som genererades kunde specifika delar av utvärderingsramverket testas. För ShipmentRouter genererades kod två gånger med lite skillnad i instruktioner, detta för att undersöka två olika implementationer för samma uppgift där olika styrkor och svagheter kunde belysas. Första genereringen innehöll duplicerad kod, där två metoder hade identisk metodkropp. För den andra genereringen så hade en av metoder väldigt många if-satser, vilket gav hög cyklomatisk komplexitet. Dock var dessa if-satser inte nästlade. Gemensamt för båda implementationerna var att projektet bestod av en C#-klass som beräknar fraktavgifter utifrån region, vikt och olika tilläggstjänster samt genererar fraktetiketter för utskrift.

Valet av just dessa projekt grundar sig i flera överväganden. För det första eftersträvades en bred spridning vad gäller projekttyp och teknikstack, eftersom ramverket är utformat för fullstackapplikationer som kombinerar .NET backend med JavaScript- respektive TypeScript-baserad frontend. Genom att inkludera projekt med både C#-kod och

JavaScript-baserad kod kunde ramverket prövas mot de språk och strukturer det är avsett att hantera. För det andra valdes projekt som täcker olika domäner och användningsområden för att säkerställa att valideringen inte begränsades till en enskild typ av applikation. För det tredje togs hänsyn till tillgänglighet och sekretess, då samtliga projekt antingen är open source eller har gjorts tillgängliga för studien utan sekretessbegränsningar, vilket möjliggör transparent redovisning av resultatet.

5. Data

5.1 Enskilda intervjuer

Som påvisat i avsnitt 3.2.4 är kodkvalitet kontextberoende. För att få en bild av kontexten på Decerno, i syfte att se till att vårt föreslagna ramverk var relevant för verksamheten, har nio utvecklare intervjuats. Intervjuobjekten representerar utvecklare som arbetar med både frontend, backend, fullstack och systemarkitektur, och i genomsnitt har de ungefär 10,7 års arbetslivserfarenhet som utvecklare. En översikt över alla intervjuobjekts respektive arbetslivserfarenhet i år, samt vilket projekt de konsultar inom, presenteras i tabell 1.

Tabell 1. Sammanställning av intervjuobjektens respektive arbetslivserfarenhet i år, samt vilket projekt de konsultar inom.

Intervjuobjekt	**Arbetslivserfarenhet (år)**	**Ingår i projekt**
Intervjuobjekt 1	7	A
Intervjuobjekt 2	8	B
Intervjuobjekt 3	20	B
Intervjuobjekt 4	16	C
Intervjuobjekt 5	14	D
Intervjuobjekt 6	16	A
Intervjuobjekt 7	14	D
Intervjuobjekt 8	0,5	E
Intervjuobjekt 9	0,5	C
Genomsnitt	*10,7 år*	

Gemensamt för alla utvecklarna är att de lyfte läsbarhet som en viktig aspekt av god kodkvalitet. I begreppet läsbarhet räknar intervjuobjekt 1, 3, 4, 5, 6 och 8 saker som att sätta lämpliga funktions- och variabelnamn, och inte blanda eller översätta svenska och engelska begrepp på konstigt sätt. Intervjuobjekt 3 och 5 beskriver att de i sina respektive team försöker att sätta gemensamma standarder för syntax och kodformatering, och delvis använder verktyg för detta. Även intervjuperson 4, 7 och 8 belyser vikten av att ha ett enhetligt arbetssätt för att kodbasen ska bli enhetlig, särskilt då de är flera personer som arbetar i den kontinuerligt. De beskriver att detta gör att utvecklarna kan känna igen sig när de rör sig mellan projekten och möjliggör att kopiera delar från ett projekt till ett annat.

Intervjuobjekt 5 beskriver att läsbarheten är extra viktig eftersom Decerno har flera kunder i offentlig sektor. Därför har flera av deras projekt sin grund i offentliga

upphandlingar, vilket gör att det varierar vem som ska arbeta med koden. Ofta tar Decerno över projekt, men ibland måste de också lämna ifrån sig projekt till andra aktörer. Vidare förklarar intervjuobjekt 5 att det bästa är när koden är tillräckligt självförklarande för att inte behöva kommentarer, men i de fall det inte går får kommentarer gärna finnas med. Även intervjuobjekt 8 och 9 lyfter att de uppskattar när koden är självförklarande och kommentarer inte behövs. Intervjuobjekt 2 diskuterar också att korrekt namngivning kan bidra till att koden blir självförklarande, och därigenom blir som ett mått för begriplighet vid pull requests. Samtidigt beskriver intervjuobjekt 2 att det vid arbete med pull requests är lätt att fastna i dessa detaljer, men funderar på hur viktigt det egentligen är i dagens utvecklingsmiljö där mer och mer AI är integrerad i utvecklingsprocesser. AI kan potentiellt förstå kontexten ändå, utan exakt korrekt namngivning.

Intervjuobjekt 3 beskriver att det är svårt när man implementerar nya riktlinjer och standarder för hur teamet ska skriva kod, eftersom det medför att man skriver likadant i de gamla kodbaserna också. Det gör att det blir svårläst när strukturen ofta växlar. Han förklarar att övergångsfaser är svåra i stora projekt eftersom man sällan tar bort kod utan lägger till nya funktioner. Det gör att kodbaserna växer med tiden, och äldre kodbaser blir lätt väldigt stora. Han beskriver att det kan kännas som att byta värld när man byter fil. Även intervjuobjekt 6 och 8 beskriver att så kallad död kod, det vill säga kod som aldrig används eller anropas, får en negativ effekt på läsbarheten och förståelsen för kodflödet, och lägger därför extra fokus på att sträva efter att hitta och ta bort död kod vid pull requests.

Intervjuobjekt 1, 2, 4, 5, 8 och 9 pratar också om kodkvalitet i termer av underhållbarhet. Intervjuobjekt 2 och 5 beskriver att det är viktigt att man lätt kan utveckla nya funktioner och förändra kodens beteende över tid för att kod ska hålla god kvalitet. Intervjuobjekt 4 menar att en egenskap för kod av bra kvalitet är att det inte ska kännas farligt eller otäckt att göra ändringar i koden. En bra arkitektur gör det billigt och enkelt att ändra systemet. Den minskar risken att små ändringar blir dyra, och gör att enskilda delar kan bytas ut utan stora konsekvenser. Dålig kvalitet på arkitekturnivå kan alltså bli mer kostsam än bristande kvalitet i en enskild fil. Intervjuobjekt 9 lyfter vikten av att ha en tydlig mappstruktur och inte för långa filer eftersom hon upplever att det gör det svårt att felsöka. Intervjuobjekt 1 och 4 beskriver att utvecklare under många år har strävat efter att komprimera kod så mycket som möjligt, och att maximal återanvändbarhet i allmänhet har setts som något bra. De menar dock att detta kan göra att man lätt blir inlåst i arkitekturen och att det kan försvåra möjligheten att göra förändringar, exempelvis om man nyttjar importerade bibliotek som slutar uppdateras och blir inkompatibla med resterande arkitektur.

Intervjuobjekt 1 anser att god kodkvalitet också handlar om huruvida koden är ändamålsenlig, det vill säga lagom komplex i förhållande till uppgiften som ska genomföras. På liknande sätt beskriver intervjuperson 4 och 8 att det är kod som åstadkommer det man vill och uppfyller sitt syfte. Intervjuobjekt 3 lyfter att bra kod ska passa in i den befintliga kodbasen utifrån aspekter som flödet i koden och hur och var data hämtas och lagras.

Intervjuobjekt 4 beskriver att man gärna har så låg koppling som möjligt för att göra komponenter mer fristående. Även intervjuobjekt 5, 6, 8 och 9 menar att det är bra att bryta ut återkommande variabler, funktioner och metoder. Intervjuobjekt 2 och 8 förklarar att man behöver hitta en balans i hur små moduler ska vara. Man vill att varje modul ska vara väl separerad, så att en funktion eller metod endast gör en sak. Samtidigt vill man

undvika hög koppling och att små moduler anropar varandra i en lång kedja. Intervjuperson 3 beskriver att fördelen med att filer och metoder är små är att de blir mer testbara. Samtidigt belyser han att små metoder kan göra det svårt att se hur alla delar i arkitekturen hänger ihop. Intervjuobjekt 1 anser att testbar är en av de viktigaste egenskaperna för god kodkvalitet, men förklarar att testbarheten är det svåraste att bedöma i en pull request. Intervjuperson 3, 5, 7 och 9 brukar alla dra ned den nya grenen för att testa att funktioner fungerar som tänkt i de fall de anser att det är svårt att avgöra genom att enbart läsa koden. Intervjuperson 7 lyfter även aspekten att för långa metoder gör det svårt att få en bra överblick som utvecklare och att risken för att fel uppstår därigenom ökar.

Andra strategier som utvecklarna har för att kontrollera kodkvalitet vid pull requests är att inte bara titta på den enskilda ändringen utan också öppna upp koden och försöka se helheten, samt att prata med den som utvecklat koden om osäkerheter uppstår. Samtidigt belyser intervjuperson 3 att detta är ett område där AI-stödda verktyg som granskar kod kan vara värdefullt eftersom den klarar av att gå igenom många fler filer än en människa. Intervjuperson 1 beskriver att även om kodgranskning vid pull requests är ett viktigt verktyg finns alltid vetskapen om att man jobbar mot en deadline. Av den anledningen brukar intervjuperson 1 försöka dela upp koden enligt aspekterna ”must have” och ”nice to have”. Med det menar han att göra en avgränsning mellan vilka delar som är ett måste att förbättra för att koden ska hålla tillräckligt god kvalitet och vad som kan justeras beroende på tidstillgång. Intervjuobjekt 4 beskriver att han vid pull requests bland annat fokuserar på att försöka se om koden löser ett problem på ett effektivt sätt i form av mängd och enhetlighet. Det gör även intervjuperson 3, men han betonar mer att han försöker se om den nya koden passar in i den befintliga arkitekturen samt att det inte finns fler, alternativa sätt att förbättra koden. Han beskriver att det är svårt eftersom det är en utmaning att se hur koden förhåller sig till andra filer som inte är inkluderade i pull requesten. Flera av utvecklarna beskriver att de överlag brukar titta på om koden ser ”snygg” ut och passar in i den övriga kodbasen. Det kan exempelvis handla om huruvida koden följer projektets etablerade struktur och uttryckssätt.

Intervjuobjekt 3 upplever att det finns en skillnad i hur utvecklare idag kan granska sin egen kod, eftersom de kontinuerligt kan få inputs och kommentarer från en AI-reviewer under processens gång. Det gör det lättare att upptäcka mindre fel som annars hade behövt granskas mer ingående av en kollega i samband med en pull request. I teamet som intervjuobjekt 3 och 9 ingår i har de därtill tillgång till en AI-granskare som ger förslag på vad man som mänsklig utvärderare bör granska vid en pull request. Viktigt att poängtera är dock att AI-granskaren saknar behörighet att själv godkänna eller neka ändringar. Intervjuobjekt 4 menar att AI har frigjort mer tid för diskussioner kring kodkvalitet. Han anser att diskussioner kopplade till risk och strategi ur ett helhetsperspektiv är viktigare än exempelvis syntax, eftersom kod i dag snabbt kan genereras med hjälp av AI. Detaljer, såsom hur knappar ska se ut, blir därmed mindre viktiga att fokusera på, eftersom de kräver mindre utvecklingstid och är snabba att ändra. Intervjuobjekt 7 och 9 förklarar att de i sina respektive team har tillgång till ett verktyg i form av ett skript som innehåller arkitekturtester. Det testar saker som korrekt namngivning, säkerställer att projekten refererar till varandra korrekt, att det inte uppstår felmeddelanden och hjälper till med formatering.

Samtliga intervjuobjekt belyste vikten av att säkerställa god kvalitet och att kodgranskning vid pull requests är ett bra tillvägagångssätt för att uppnå just god kvalitet i ett team. Trots det var det ingen av intervjuobjekten som beskrev att de har ett

systematiskt tillvägagångssätt för att genomföra kodgranskning av pull requests, eller att de använder sig av algoritmiska verktyg. Flera av utvecklarna beskriver att de har jobbat tillräckligt många år för att kunna upptäcka brister enbart genom att läsa koden, och i de fall de är osäkra tar de, som tidigare beskrivit, ned grenen och testkör denna. Samtidigt uttrycker vissa av intervjuobjekten att de i början hade svårt att veta hur de skulle tänka vid pull requests. Att kunna genomföra en god kodgranskning verkar därmed bygga mycket på att utföraren är senior och kan läsa sig till funktionalitet och buggar. De får en känsla för koden som bygger på erfarenhet. Flera av utvecklarna beskriver att det har skett ett skifte i hur mycket tid man lägger på att skriva respektive granska kod. Om man tidigare la majoriteten av tiden på att själv skriva kod och en mindre andel på att granska, ser situationen ibland tvärt om ut idag, och man lägger relativt sett mer tid på kodgranskning idag än tidigare.

I vissa av projekten finns AI-granskare som stöd vid pull requests, och i de fall utvecklare har tillgång till detta beskrivs det som värdefullt att få input i vad man ska kolla efter. Intervjuobjekt 4 beskriver att han hade önskat att det fanns ett integrerat statiskt verktyg i samband med pull requests där utvecklarna kunde följa mått över tid. Han efterfrågar exempelvis ett verktyg där man kan se att den cyklomatiska komplexiteten är konstant, och beskriver att om den då plötsligt skulle dra i väg vet man att man behöver skruva något. Han beskriver att ju mindre vi läser koden, desto mer behöver vi olika sätt för att mäta koden för att kunna förstå den. Det tror han kan bidra till att man jobbar mer med helheten istället för att man stirrar sig blind på små detaljer och missar de verkliga kostnaderna.

5.2 Gruppdiskussion AI first

På Decerno finns ett team som arbetar i ett projekt med uttalad strategi att använda AI i så stor utsträckning som möjligt. Projektet uppstod genom att man i anbudet inför en upphandling inkluderade ett avsnitt om att arbeta ”AI first”, vilket bidrog till att sänka priset mot kunden. Syftet med detta arbetssätt är att utforska olika AI-verktyg och arbetsmetoder för att förbli konkurrenskraftiga i en föränderlig omvärld, där mjukvaruutveckling i allt högre grad präglas av AI. Projektets arbetssätt är med andra ord helt drivet av företagets ledningsgrupp. Kunden i fråga har ingen särskild preferens vad gäller arbetssätt. Teamet består av sju personer, varav sex deltog i diskussionen, se tabell 2.

Tabell 2. Sammanställning av intervjuobjektens respektive roll i AI first teamet och arbetslivserfarenhet i år.

Namn	Roll	Erfarenhet (år)
Intervjuobjekt 10	Utvecklare	2-3
Intervjuobjekt 11	Tech Lead	16
Intervjuobjekt 12	Utvecklare	22
Intervjuobjekt 13	Projektledare	10-15
Intervjuobjekt 14	UX (utbildad utvecklare)	2
Intervjuobjekt 15	Driver området ”way of working”	26

I praktiken innebär AI first i detta projekt att utvecklarna vibe-kodar, vilket innebär att de skriver promptar till LLM:er som utifrån dessa genererar kod. Det finns inga särskilda

riktlinjer för vilka verktyg som ska användas, utan det avgörs av den enskilde utvecklaren eller UX:aren. I vissa delar av processen valde man dock att inte använda AI. Det gällde dels när man skulle sätta upp infrastrukturen i produktionsmiljö, eftersom säkerheten där är särskilt känslig och att det är dyrt att göra infrastrukturella ändringar i efterhand. Därtill valde man att manuellt, tillsammans med kunden, skapa features från de epics som kunden tillhandahöll, i syfte att säkerställa att dessa låg i linje med kundens kravställning. Från varje feature användes sedan AI för att bryta ned problemen i krav och mindre beståndsdelar för utvecklarna att börja arbeta med. Teammedlemmarna beskriver vidare att projektet inte handlar om att använda AI för att få tips och tricks eller små funktioner, utan att snarare om att generera hela features utifrån kraven. Samtliga utvecklare i teamet beskriver att de inte har skrivit en hel rad kod sedan projektet inleddes och förklarar att deras arbete handlar mer om att skriva enstaka ord och ta bort överflödig kod. De använder en AI-reviewer för att granska koden vid pull requests, men i slutändan godkänns eller nekas ändringarna manuellt av en människa.

Teamets upplevelse är att den genererade backendkoden generellt är mer välstrukturerad än frontendkoden. Utvecklarna beskriver att de i frontendkoden oftare upplever att ett problem löses på ett konsekvent sätt på flera ställen i koden, för att sedan plötsligt få en helt annan lösning på andra ställen. Dessutom upplever de att AI skriver långa och komplicerade metoder. Sammantaget väcker detta frågor om huruvida det kan leda till buggar och problem längre fram. De beskriver också att de inte vet om det spelar någon roll eftersom agenterna enkelt verkar kunna åtgärda dessa problem. De tror att en möjlig förklaring till att backendkoden är mer välstrukturerad än frontendkoden är att det finns tydligare regler uppsatt för backendarkitekturen, som LLM:erna får tillgång till. En annan möjlig förklaring som de diskuterar är att man vid frontendutveckling ser UI:et först, vilket gör att koden bakom kan vara sämre strukturerad än vad den först verkar, till skillnad från backendutveckling där utvecklaren ser och bedömer koden först. När det gäller samarbete och att få de olika delarna att vara kompatibla med varandra har teamet inte upplevt några svårigheter. Tvärtom upplever utvecklarna att den kod som genereras utifrån UX:arens prototyp i Lovable fungerar förvånansvärt bra.

Utvecklarna beskriver att de lägger mer tid på kodgranskning i form av att granska koden som man själv producerat, men granskning av andras kod är inget man lägger mer tid på. De förklarar att det finns en känsla av osäkerhet när så stora mängder kod genereras. De upplever att volymen kod blir lätt väldigt stor och att de därför inte har tid att manuellt granska all kod. Därför förlitar dom sig mycket på AI-reviewern vid pull request. Samtidigt resonerar de kring att deras aktiva arbete med AI-reviewern i praktiken kan bidra till att säkerställa kvaliteten i högre grad än om koden hade skrivits manuellt utan motsvarande granskningsstöd. De menar också att det kanske inte längre är rimligt att fullt ut förstå all kod som produceras, och betonar att det finns en skillnad mellan hur AI-agenter genererar kod och hur mänskliga utvecklare traditionellt skriver den. Människor eftersträvar i högre grad att skriva mer kompakt och kortare kod.

Utvecklarna beskriver att de inte upplever några större hallucinationer eller att kvaliteten på koden blir sämre när den genereras med hjälp av AI. De upplever att AI hjälper till att sudda ut flaskhalsar och ger projektet en mer jämn kapacitet, eftersom det är lättare att ta tag i saker själv och på det sättet blir varje person mer autonom. En fråga som däremot kom upp från flera håll i teamet var hur man ska hantera alla ”extra” förslag som AI:n kommer med. Det kan smyga sig in sådant som framstår som rimligt inom ramen för en uppgift, men som ingen människa faktiskt har bett om. På ett sätt är detta inte unikt eftersom utvecklare också kan föreslå eller lägga till något extra, men det verkar uppstå

oftare i arbetet med AI. Därför efterfrågas ett sätt att få bättre kontroll över vilka extra saker som är motiverade att genomföra och inte.

Teammedlemmarna beskriver att mycket av kvalitetssäkringen är kundens ansvar, men de förklarar att de själva tar buggrapporter och genomför automatiserade tester för att undersöka kvaliteten, samt att en pilot planeras att genomföras innan lansering. De upplever att de blir lite lidande av att de inte har en person som är ansvarig för test. Ibland har kunden upptäckt saker teamet borde ha fångat tidigare. Samtidigt ingick det i riktlinjerna de fick, att de skulle testa att genomföra projektet utan en dedikerad testansvarig. För att ändå känna sig säkra på att det inte förekommer säkerhetsbrister i produkten de utvecklar finns det ett team som bistår med säkerhetsgranskning vid sidan av projektet som Decerno står för budgetmässigt.

Framåt tror utvecklarna att man kan jobba vidare och förvalta projektet både med AI first-metodik och med mer traditionella metoder. De förutspår att förvaltningen i vissa avseenden kan bli något enklare än i mer traditionella projekt, eftersom de upplever att koden övergripande har en tydlig och konsekvent struktur. Samtidigt lyfter de fram vissa potentiella utmaningar. En utmaning är att kodbasen blir större snabbare vid AI first-metodik kontra traditionell mjukvaruutveckling. De menar också att de ser att det, på detaljnivå, förekommer inkonsekvens i hur problem löses. Samma typ av problem implementeras i de flesta fall på likartat sätt, men får i vissa fall en avvikande lösning. Det skapar inkonsekvens i kodens utformning, trots att den övergripande strukturen samtidigt upplevs som tydlig och konsekvent. Utvecklarna påpekar dock att det i nuläget är osäkert huruvida detta kommer att utgöra ett praktiskt problem i framtiden eller inte.

Vidare diskuterar teammedlemmarna möjligheten att skala upp AI first-projektet. De menar att metodiken leder till att man göra stora ändringar ofta och att mycket information utväxlas mellan medlemmarna i teamet. Detta menar dom kan leda till att det är svårt att skala upp teamstorleken. De menar att man istället skulle behöva skala i kalendertid om man vill öka projektets storlek.

Med det sagt tror de att det finns goda chanser för AI first-metodiken att bli standard inom mjukvaruutveckling förutsatt att AI-modellerna fortsätter att utvecklas och bli bättre. Om modellernas utveckling inte stagnerar, utan de fortsätter att förbättras, menar vissa teammedlemmar att en återgång till mer traditionella metoder kan vara osannolik. Till exempel beskriver en medarbetare som tidigare varit skeptisk till projektets arbetsmetodik att han nu ändrat sig helt. Han förklarar att han nog skulle känna sig bestulen på någonting om man skulle gå tillbaka och inte få använda AI i framtida projekt. Ett ytterligare exempel är att flera av medarbetarna beskriver att det har skett ett skifte där en agent blir ens närmsta kollega och den man frågar i första hand för att komma vidare i sitt arbete.

5.3 Avgränsningar i den algoritmiska kodgranskningen

I avsnitt 4.4 presenterades valet av vilka algoritmiska mått som används i vår kodgransking för att mäta kodkvalitet. Innan dessa mått kan tillämpas måste det fastställas vilka delar av en kodbasen som faktiskt utgör meningsfulla analysobjekt. Det är inte metodologiskt motiverat att analysera samtliga filer i en kodbas med algoritmiska mått, eftersom alla filer inte representerar källkod med ett innehåll som är meningsfullt att bedöma ur ett kvalitetsperspektiv. Vissa filer innehåller främst deklarationer av datastrukturer eller konfigurationsmönster snarare än faktisk exekverbar programlogik.

Andra filer är automatgenererade, exempelvis kompilerad eller transformerad kod som skapas av byggverktyg, ramverk eller pakethanterare. Det finns också filer som utgör distributionsfärdiga artefakter, det vill säga kod som redan bearbetats för att kunna köras i produktionsmiljö, exempelvis bundlad, transpilerad eller minifierad kod. Gemensamt för dessa filer är att de inte utgör mänskligt skriven källkod och att de därför har ett begränsat värde i en analys som syftar till att bedöma struktur, komplexitet och underhållbarhet. För att öka validiteten i analysen görs därför ett inledande urval där vissa typer av filer exkluderas från att undersökas med de algoritmiska måtten.

I kodbaser där Entity Framework används är det vanligt att delar av databasabstraktionen modelleras genom klasser som ärver från DbContext. Dessa klasser representerar kopplingen mellan applikationen och databasen och följer ett ramverksstyrt mönster där properties motsvarar databastabeller och särskilda metoder används för konfiguration av datamodellen. Strukturen i sådana klasser skiljer sig därför från vanliga klasser med exekverbar programlogik. Låga eller avvikande värden på de algoritmiska måtten i en DbContext-klass behöver därmed inte indikera bristande kodkvalitet, utan kan snarare vara en konsekvens av hur ramverket kräver att klassen utformas. Av detta skäl exkluderas klasser som ärver från DbContext från den algoritmiska granskningen.

Vidare exkluderas filer vars Halstead-volym är lika med noll. Om värdet är noll innebär det att filen saknar den typ av exekverbara uttryck som de algoritmiska måtten bygger på. Dessa filer kan vara viktiga för systemets arkitektur, men är inte meningsfulla att inkludera i en algoritmisk kodgranskning.

Även minifierade filer exkluderas från analysen. Minifierad kod är kod som automatiskt har skrivits om med hjälp av byggverktyg för att minska filstorleken. Den utgör därmed inte en rättvis representation av utvecklarens ursprungliga källkod. Om minifierade filer inkluderas i analysen leder det till förvrängda värden. Därför betraktas de inte som relevanta för kodanalys.

På motsvarande sätt exkluderas mappar som innehåller genererat material, byggartefakter eller externa beroenden, eftersom dessa inte utgör projektets egentliga källkod och därför inte bör ingå i analysen. Det gäller exempelvis mappar såsom node_modules, dist och build i JavaScript- och Typescript-projekt samt packages, nuget, bin och obj i C#-projekt. Dessa mappar har olika funktioner men gemensamt är att deras innehåll skapas automatiskt utifrån källkoden eller hämtas in som tredjepartsberoenden. Node_modules, packages och nuget används för att lagra de externa bibliotek som laddas ner till utvecklingsmiljön. Build och dist innehåller en bearbetad version av källkoden som är anpassad för att göra applikationen mer effektiv vid körning och förbättra laddningstid. Bin och obj innehåller kompilerad kod och andra byggartefakter som genereras under byggprocessen i C#-projekt. Gemensamt för dessa mappar är att innehållet inte är utvecklat för att underhållas på samma sätt som projektets ursprungliga källkod. Om dessa filer inkluderas i analysen finns risk att samma funktionalitet analyseras flera gånger, dels i sin ursprungliga form, dels i en genererad eller bearbetad version.

Även testfiler exkluderas från den algoritmiska kodgranskningen. De mått som används i denna studie är huvudsakligen utvecklade för produktionskod och är i första hand avsedda att fånga kvalitetsaspekter i kod som implementerar systemets faktiska funktionalitet. Testkod följer ofta andra designideal än produktionskod, där exempelvis tydlighet och repetition kan vara funktionellt motiverade för att göra testfallen lättare att förstå, isolera och underhålla (Agarwal, 2025). Strukturer som i produktionskod skulle

kunna tolkas som låg kvalitet behöver därför inte ha samma innebörd i testkod. Om testfiler inkluderas i analysen finns en risk att de algoritmiska måtten ger missvisande värden och därmed försvårar tolkningen av resultatet. För att säkerställa att analysen avser den kod som faktiskt implementerar systemets funktionalitet exkluderas därför testfiler från den fortsatta granskningen.

6. Resultat

I detta avsnitt presenteras resultatet från valideringen av det utvecklade utvärderingsramverket. Resultatet redovisas per kodsegment, där utvecklarnas iakttagelser från den manuella kodgranskningen jämförs med de bedömningar som genererats av utvärderingsramverket. Totalt analyserades fyra kodsegment hämtade från projekten RealWorld, Maya och ShipmentRouter, vilka beskrevs närmare i avsnitt 4.9. För varje kodsegment redovisas först utvecklarnas synpunkter och därefter utvärderingsramverkets utfall, i syfte att tydliggöra var de båda granskningsperspektiven sammanfaller respektive skiljer sig åt.

Det första kodsegment som utvärderades tillhör projektet RealWorld. För detta projekt påpekade två av utvecklarna valet av variabelnamn för att kolla om en användare var författare. De menar att flera personer potentiellt skulle kunna ha samma namn och att mäta mot ett UserID skulle kunna vara mer lämpligt än användarnamn. En utvecklare var osäker på om metoden AssertOwnerShip returnerar forbid som den ska enligt taskbeskrivningen. En annan utvecklare påpekade viss duplicering i de två första metoderna. I övrigt hade utvecklarna inga synpunkter på koden**.**

Vad gäller det utvecklade utvärderingsramverket genomfördes sammanhållningsanalys på en fil. Resultatet visar att TCC uppgår till 100 procent och att LCOM1 är 0, vilket enligt AI-reviewern indikerar mycket hög sammanhållning. Bedömningen motiveras med att filens metoder använder samma attribut, vilket tyder på att de är samlade kring ett gemensamt ansvarsområde. Detta bedöms av AI-reviewern vara ändamålsenligt i den aktuella filen eftersom den utgör en auktoriseringstjänst vars metoder förväntas samverka kring åtkomst till databasen och kontroll av den aktuella användaren. Vad gäller underhållbarhetsindex analyserar ramverket tre filer och beskriver vidare att två av dessa avviker från resten av kodbasen. Båda dessa filer har lägre underhållbarhetsindex än kodbasens genomsnitt och förklarar att det beror på att filerna har nästan dubbelt så många rader källkod som kodbasens genomsnitt. Trots det bedömer AI-reviewern i ramverket att implementationen är rättfram och därför inte behöver åtgärdas vidare. I analysen av cyklomatisk komplexitet lyfter ramverket tre metoder som alla rekommenderas av AI-reviewern att behållas i sin nuvarande form. Detta motiveras med att deras cyklomatiska komplexitet 3 är låg och motiveras av enkel, tydlig och testbar logik utan onödig förgrening.

Utvärderingsramverket hittade ingen kodduplicering i de genererade filerna. Däremot identifierades brister som delvis strider mot kravspecifikationen, kopplat till användning av den implementerade auktoriseringslogiken. Ramverket beskriver att service-klassen inte anropas av några endpoints, eftersom edit- och delete-endpoints för artiklar ännu inte finns implementerade i controller-klassen. Det är dock viktigt att notera att controller-klassen inte genererades i detta steg utan redan existerade i kodbasen sedan tidigare, och därmed inte inkluderades i filurvalet som utvecklarna granskade. AI-reviewern

klassificerar därför implementationen som ofullständigt integrerad snarare än felaktig, där den utvecklade logiken för ägarskapskontroll i nuläget saknar koppling till framtida endpoints för redigering och borttagning av artiklar. Möjligt är att dessa kan komma att läggas till i senare steg i kodgenereringen för projektet.

Det andra kodsegmentet tillhör Maya-projektet. För detta segment var utvecklarnas åsikter mer varierande. Alla tre utvecklarna beskrev att koden var enkel att förstå och att den såg ut som React-kod brukar göra. Samtidigt ansåg en av de tre att koden var lite onödigt komplext skriven. Hon menade bland annat att defaultvärden sattes flera gånger vilket skapade duplicering, och föreslog att reset bör användas istället. Hon pekade även ut ett par användningar av reset samt backendanrop som inte verkade behövas, vilket skapar onödig kod. Vidare upptäckte hon ”fulfixar” där dummy state-variabler används enbart för att tvinga omrendering så att data hämtas på nytt efter varje ändring. Detta är ett tecken på att något underliggande är fel, och att problemet lösts tillfälligt på klientsidan istället för vid källan.

Utvärderingsramverket flaggade för att underhållbarhetsindexet i de genererade filerna är markant lägre än genomsnittet i kodbasen. AI-reviewern beskriver att det beror på hög cyklomatisk komplexitet och jämförelsevis stora filer med högt antal SLOC, jämfört med snittet i kodbasen. Samtidigt jämförs filerna med komponenter som fyller liknande funktioner i projektet, och bedömningen är att detta mönster återkommer i funktionsrika administrationssidor och modaler. De lägre underhållbarhetsindexet behöver därför inte tolkas som ett kvalitetsproblem, utan snarare som en följd av den inneboende komplexiteten i den här typen av komponenter. Vad gäller sammanhållning och kodduplicering hittade ramverket inget att anmärka på. Däremot identifierade AI-reviewern en avvikelse kopplad till borttagning av organisationer. Det handlar om att den genererade implementationen av DeleteOrganisation-klassen saknar kontroller för om organisationen fortfarande refereras innan borttagning genomförs. Detta bryter mot kravet att sådana organisationer inte får raderas och istället ska resultera i felstatuskod. AI-reviewern lyfter därmed att implementationen visserligen fungerar tekniskt, men att en central affärsregel inte efterlevs. DeleteOrganisation genererades i denna körning men inkluderades inte i den manuella granskningen. Detta eftersom valideringen annars hade blivit för omfattande.

Det tredje kodsegmentet var ShipmentRouter med implementation som inkluderade en duplicerad metod. Alla utvecklare upptäckte detta och menade att det var överflödigt. En utvecklare påpekade att eftersom båda metoder var identiska i form, alltså vilka argument de tog som input och vilka värden de returnerade, skulle inte backup-metoden heller fungera i de fall den “vanliga” metoden inte skulle köras. En utvecklare påpekade att backup-metoden dessutom aldrig anropades. Den tredje utvecklaren ansåg att eftersom metoden i sig har väldigt lite funktionalitet och endast anropar en annan funktion är det värt att utvärdera huruvida den ska formuleras som en egen metod över huvud taget. Två av utvecklarna beskriver också att de inte gillar hur värden för exempelvis regioner är hårdkodat utan anser att dessa borde hämtas dynamiskt för att möjliggöra ändringar och återanvändning av dessa.

Även utvärderingsramverket identifierade och flaggade den duplicerade metoden. Utöver det genomfördes en analys av sammanhållningen på en av de genererade filerna, vilket gav resultaten TCC=0,0 och LCOM1=10. Underhållbarhetsanalysen bedömde filens underhållbarhetsindex som låg, främst på grund av dess antal SLOC. Samtidigt beskrev analysen att filen har en tydlig logik, utan djupt nästlade metoder eller otydligt flöde, och

att den därför inte kräver några vidare åtgärder. Analysen av cyklomatisk komplexitet visade värdet 8 för en metod i den genererade filen. Samtidigt konstaterades att koden inte bör refaktoreras för att minska denna komplexitet, eftersom dess struktur är sekventiell snarare än nästlad. Varje villkorssats ökar den totala kostnaden lika mycket och bidrar därför till komplexiteten, men och att refaktorera skulle enligt analysen inte öka läsbarheten.

Utvärderingsramverket identifierade även tecken på over-engineering i implementationen i form av att privata statiska fält lagrades i dictionaries för att hantera regioner och viktklasser, trots att datamängden var liten och relativt statisk. Ramverket beskriver att det ökar komplexiteten utan att ge någon tydlig funktionell nytta. Som förslag på en enklare lösning ges att använda switch-uttryck eller statiska arrayer direkt i metoden, vilket hade ökat läsbarheten. Den genererade koden fungerar korrekt men lösningen är mer komplex än vad problemet egentligen kräver.

Det fjärde kodsegmentet var implementationen av ShipmentRouter med hög cyklomatisk komplexitet. Den höga cyklomatiska komplexiteten kom främst från att många if-satser staplats på varandra. Alla utvecklarna beskrev att dessa gjorde det svårt att följa flödet. En av dem menade att switch-satser vore bättre. Även för den andra implementationen upplevde utvecklarna att det var för mycket data hårdkodat i metoderna och menade att detta bör flyttas till en databas och anropas automatiskt. Andra faktorer som utvecklarna tyckte hade negativ inverkan på läsbarheten var att variabelnamnen för kostnadsberäkningarna inte var tillräckligt intuitiva och att de därför borde bytas ut.

Utvärderingsramverket flaggade även det för den höga cyklomatiska komplexiteten i flera av de genererade filerna. Bland annat beskriver ramverket att metoden som beräknar fraktkostnaden får väldigt hög komplexitet till följd av att flera oberoende villkorsstyrda kontroller görs efter varandra. Den höga cyklomatiska komplexiteten medför också att underhållbarhetsindexet blir lågt vilket utvärderingsramverket belyser. Därtill finns det några filer som i kombination med hög cyklomatisk komplexitet också innehåller relativt mycket fler SLOC. Detta medför också att underhållbarhetsindexet sjunker. Vidare görs en sammanhållningsanalys som ger resultat TCC på 0.44 och LCOM1 på 1. AI-reviewern beskriver att det är väntade resultat för klasser som innehåller affärslogik och som har i uppgift att på detta vis beräkna kostnader baserat på olika parametrar. Utvärderingsramverket hittar ingen kodduplicering bland de genererade filerna.

7. Diskussion

7.1 Resultatskillnader i litteraturstudie och intervjustudie

En generell avvikelse som visats mellan resultaten från litteraturstudien, intervjustudien och resultatvalideringen är vikten av kodkommentering. Börstler et al. (2023) menar att korrekt och genomtänkt kodkommentering kan främja läsbarheten av kod, men betonar samtidigt att för mycket kommentarer kan leda till distraktion. Även Martin et al. lyfter, som tidigare beskrivits, vikten av att istället för att kommentera kod fundera på olika sätt att omformulera eller refaktorera koden så att det blir mer intuitivt för läsaren. Eltabakh, Nabil Soudi och Shawky (2024) beskrev i sin studie att de sett en trend i att AI-genererad kod tenderar att innehålla fler kommentarer jämfört med mänskligt skriven kod. I

intervjustudien framgick att intervjudeltagarna föredrar när koden är tillräckligt självförklarande för att kommentarer inte ska behövas, men att de kan vara ett bra komplement då självförklarande kod är svårt att uppnå. Koden som genererats för valideringen innehöll inga kodkommentarer alls. Detta noterades inte av någon av utvecklarna som deltog i valideringen, trots att flera av dem ansåg att viss kod hade ett flöde som ansågs svårt att följa och förstå.

Ytterligare en skillnad som framkommit i resultaten från intervju- respektive litteraturstudien var synen på begreppen säkerhet och prestandas vara eller icke vara i kodkvalitetbegreppet. Medan ISO-standarden (ISO/IEC 25023) inkluderar båda dessa och Yetiştiren et al. (2023) inkluderar säkerhet som en aspekt i sin studie, lyftes inte detta som något centralt i hur de intervjuade deltagarna ser på kodkvalitetsbegreppet. Utvecklare 5 och 8 nämnde begreppet säkerhet, och utvecklare 2 och 8 nämnde prestanda under sina respektive intervjuer. Alla var eniga om att säkerhet och prestanda inte är vad de primärt tänker på i begreppet kodkvalitet. Det framstår snarare som att god prestanda och säkerhet anses vara grundläggande egenskaper hos kod och därmed alltid förväntas uppfyllas på Decerno. Till följd av detta exkluderades dessa aspekter från det utvecklade utvärderingsramverket eftersom det kan betraktas som en självklarhet.

Martin et al. argumenterar för att AI-stödd utveckling har rört sig bort från radvis kodgenerering mot ett arbetssätt där hela moduler genereras vilket innebär att utvecklares fokus i högre grad bör ligga på övergripande systemarkitektur. Intervjuresultaten pekar i en liknande riktning, men betonar samtidigt en syn på modularisering som innefattar mer avvägning. De framhåller att de visserligen ser ett värde i att dela upp kod i mindre, tydligt avgränsade funktioner och metoder, men betonar också att för små metoder kan leda till att det är svårare att få en överblick. Det kan också öka beroenden mellan komponenter och göra arkitekturen svårare att förstå. Därför krävs att man i praktiken balanserar att moduler ska ha ett entydigt ansvar med systemförståelse. Istället framhåller Martin et al. modularitet som en strategisk förutsättning för flexibilitet och utbytbarhet i AI-genererade system. Detta är även relevant i relation till sammanhållningsmåtten, eftersom hög sammanhållning kan tolkas som ett tecken på att en modul är samlad kring ett gemensamt ansvarsområde, men inte ensamt fångar de praktiska avvägningar som utvecklarna beskriver att man behöver göra.

7.2 Valideringsresultat

Resultaten från den manuella kodgranskningen och det utvecklade utvärderingsramverket visar både likheter och skillnader i hur kodkvalitet bedöms. Koden genererad till RealWorld-projektet väckte semantiska frågor för alla utvecklare i form av valet att använda username istället för exempelvis userID, men det var bara en av utvecklarna som ifrågasatte huruvida koden returnerade det värde som var tänkt. Dessa aspekter fångade inte AI-reviewern. Samtidigt bekräftade både utvecklare och ramverk att koden överlag var rimlig och enkel att följa.

Detta resultat tyder på att utvecklarnas utvärdering och utvärderingsramverket fångar olika typer av problem. Medan utvärderingsramverket främst är träffsäkert i bedömningen av strukturella kodegenskaper, fångas mer subtila semantiska och domännära problem samt problem kopplat till praktiska designval i större utsträckning av utvecklarna. Exempel på detta är val av variabler, att värden som enligt utvecklarna borde hämtas dynamiskt istället har hårdkodats, samt andra typer av tillfälliga lösningar. Fokus

ligger därmed i hög grad på korrekthet och robusthet i den mänskliga granskningen. Utvärderingsramverket tycks däremot vara mer träffsäkert i bedömningen av strukturella egenskaper i koden, såsom komplexitet, filstorlek och sammanhållning. Resultatet antyder alltså att ramverket främst fångar övergripande strukturella kodegenskaper, medan mer subtila semantiska problem och frågor kopplade till designval i större utsträckning uppmärksammas av mänskliga utvecklare.

För koden som genererades i Maya-projektet visar resultatet av valideringen att utvärderingsramverket fångar dess låga underhållbarhetsindex, men trots det bedömer AI-reviewern att det kan vara förväntat för den typ av kod som genererats och att det därför är acceptabelt. Detta kan tolkas som att utvärderingsramverket gör en mogen, kontextuell bedömning och undviker överreaktion på avvikande kvalitetsvärden i React-komponenter. Samtidigt kan det innebära att utvärderingsramverket riskerar att bli för tolerant och normalisera dåliga mönster i kodbasen. Utvärderingsramverket fångade inte dupliceringen av defaultvärden och onödiga anrop i samma utsträckning som utvecklarna. Det är återigen ett exempel på något som utvecklarna anser vara ett problem men som utvärderingsramverket antingen inte upptäcker alternativt inte anser skapa tillräckligt stora strukturella problem att det är värt att flagga för.

Vidare beskrev utvecklarna att koden i Maya-projektet ser ut som React-kod ”brukar” göra. Detta är intressant i relation till Maikantis et al. (2024), som undersöker sambandet mellan kodkvalitet och kodens estetiska uttryck. I deras studie bygger det empiriska underlaget på projekt som återanvänts från Nikolaidis et al. (2023), det vill säga från en period då AI-genererad kod ännu inte var allmänt förekommande. Projekten valdes dessutom ut utifrån kriterier som omfattande utvecklingshistorisk med strukturerad commit-process, för att säkerställa att projekten representerar mogna och väletablerade kodbaser (Maikantis et al., 2024, s. 1, 9–10). Därför är det rimligt att anta att den kod som studien bygger på huvudsakligen har utvecklats av mänskliga programmerare.

Mot denna bakgrund blir studiens resultat intressant i dagens kontext, där en växande andel kod helt eller delvis genererats med hjälp av AI-baserade verktyg. Om AI-genererad kod ofta följer etablerade stilistiska mönster och därför ser välstrukturerad och korrekt ut vid en första anblick, är det inte säkert att sambandet mellan kodens estetiska uttryck och dess faktiska kvalitet ser ut på samma sätt som i tidigare studier. I en sådan kontext kan kodens utseende ge intryck av hög kvalitet, utan att detta nödvändigtvis motsvaras av exempelvis god underhållbarhet eller korrekthet. Samtidigt tyder resultaten från vår genomförda intervjustudie på att kodens estetik spelar roll i utvecklarnas bedömning, huruvida den framstår som bekant och följer förväntade uttryckssätt. Flera av de intervjuade utvecklarna förklarar att de vid pull requests tittar på om koden ser ”snygg” ut, exempelvis om syntax följer etablerade kodstandarder. Det skulle vid valideringen kunna innebära att kod som vid första anblick ser välstrukturerad och bekant ut också uppfattas som mer kvalitativ, även om den i praktiken kan innehålla problem som är svårupptäckta.

För de kodsegment som genererades två gånger i ShipmentRouter-projektet indikerar resultaten att utvärderingsramverket fungerar väl i linje med sin design. Den algoritmiska analysen identifierade både kodduplicering och hög cyklomatisk komplexitet, och AI-reviewern flaggar för dessa. Både den mänskliga granskningen och granskningen genomförd av utvärderingsramverket markerade dessutom om kravspecifikationen följs, det vill säga om den funktionalitet som beskrevs i taskbeskrivningen faktiskt hade implementerats.

Utifrån detta framstår det som att de två utvärderingsmetoderna, den mänskliga kodgranskningen och utvärderingsramverket, delar centrala bedömningsgrunder. Gemensamt för dem är att de lägger vikt vid om kodens funktionalitet motsvarar kravspecifikationen, samt att de upptäcker kodduplicering och hög cyklomatisk komplexitet. Detta ligger också i linje med hur flera deltagare i intervjustudien beskrev sitt tillvägagångssätt när de genomför kodgranskning, där kontroll av att uppgiften faktiskt har lösts utgjorde en viktig del. Resultaten av valideringen antyder därmed att ramverket i dessa avseenden fångar aspekter som också är centrala i mänsklig kodgransking.

Vidare visar resultaten av valideringen att utvärderingsramverket fungerar väl för att upptäcka klassiska kodkvalitetsproblem såsom hög cyklomatisk komplexitet, lågt underhållbarhetsindex och kodduplicering, samt att ramverket kan sätta dessa mätvärden i förhållande till kodbasens normer och genomsnitt. Det gör att analysen blir mer kontextkänslig och ger mer insikt än att bara exempelvis titta på absoluta tröskelvärden. Samtidigt finns det därmed också en risk att problem blir relativa, eftersom en dålig kodbas kan göra att ny, svag kod anses vara normal.

Något som genomgående framträder vid valideringen är att AI-reviewern vid samtliga kodsegment visar en tydlig benägenhet att motivera dåliga mätvärden snarare än att flagga dem som problem. I RealWorld förklaras avvikande underhållbarhet med högre SLOC och i Maya med att liknande komplexitet återfinns i andra komponenter i kodbasen. ShipmentRouter-segmentet med duplicerad kod bedöms ha lågt underhållbarhetsindex på grund av filens storlek, men eftersom logiken har ett tydligt flöde anses koden ändå inte behöva underkännas. För det andra ShipmentRouter-segmentet, som har hög cyklomatisk komplexitet, beskrivs det låga underhållbarhetsindexet som naturligt för klasser med affärslogik av denna typ. Sammantaget behöver detta i sig inte utgöra ett problem om de motiveringar AI-reviewern ger faktiskt är välgrundade. Om AI-reviewern däremot ursäktar verkliga kvalitetsproblem med kontextuella motiveringar finns en risk att utvärderingen istället leder till falska negativa bedömningar, där bristfälliga mönster i kodbasen accepteras snarare än lyfts fram.

En möjlig förklaring till detta beteende är att AI-reviewern designats att hellre fria än fälla vid osäkerhet, vilket är ett medvetet val för att undvika falska positiva resultat. Samtidigt kan denna benägenhet också kopplas till ett välkänt fenomen hos språkmodellerna, ofta benämnt ”sycophancy”. Det innebär att modellerna har en tendens att producera svar som överensstämmer med användarens förväntningar eller den input de fått, snarare än att leverera kritiska eller motsägande bedömningar (Tan, 2026). Denna egenskap är delvis ett resultat av hur modellerna tränas, där hjälpsamma och samarbetsvilliga svar ofta belönas. Att AI-reviewern eventuellt lider av denna benägenhet är en viktig aspekt att ha i åtanke vid användning av utvärderingsramverket, och det vore värdefullt att undersöka närmare, både för att bättre förstå när kontextuella motiveringar är välgrundade och när de utgör felaktiga bortförklaringar. Det bör dock noteras att förmågan att styra språkmodeller till att minska denna benägenhet ligger utanför ramen för denna uppsats, eftersom problemet i grunden kommer från LLM:ernas natur och därför är svårt att fullt ut kontrollera genom prompt-design.

7.3 Determinism och AI-baserad bedömning

En av de mest grundläggande utmaningarna med att basera utvärderingsramverket till stor del på en AI-reviewer är att språkmodeller inte är deterministiska. Två identiska promptar till samma modell ger två olika resultat, vilket står i direkt motsättning till det krav på determinism som ställdes upp för utvärderingsprocessen redan i kravspecifikationen. Där fastslås att utvärderingsmetoden ska generera så identiska resultat som möjligt vid upprepade körningar. Detta gör determinism till en av de svåraste parametrarna att hantera i utvärderingsramverkets utformning och flera designval har gjorts i syfte att uppfylla kravet, trots att det slutgiltiga omdömet fattas av en AI.

Ett centralt tillvägagångssätt för att öka determinismen är att låta AI-reviewern utgå från algoritmiska kodkvalitetsmått, vilka i sig är fullt deterministiska. Genom att ge AI-reviewern dessa mått som input får modellen ett stabilt underlag att förhålla sig till, vilket kan antas minska variationen mellan körningar. Även om AI-reviewerns slutgiltiga omdöme fortfarande är beroende av modellens icke-deterministiska natur, förankras bedömningen i en deterministisk grund som inte förändras mellan körningar. På så sätt blir utvärderingen mer förutsägbar än om AI-reviewern enbart hade förlitat sig på en fri tolkning av koden.

I detta sammanhang uppstår en intressant avvägning mellan hårda och mjuka tröskelvärden för de algoritmiska måtten. Hårda tröskelvärden skulle göra utvärderingen ytterligare mer deterministisk, eftersom ett mätvärde över eller under ett fastställt gränsvärde alltid skulle leda till samma utfall, oberoende av AI-reviewerns tolkning. Detta skulle dock samtidigt innebära en risk för fler falska positiva resultat, eftersom kod som överstiger ett tröskelvärde av legitima skäl då skulle underkännas trots att den i sin kontext är välmotiverad. Detta knyter an till den kritik av kvantitativa kodmått som lyfts av exempelvis Oliveira et al. (2021), som menar att fasta trösklar ofta blir missvisande eftersom vad som utgör ett högt eller lågt värde varierar beroende på kontext speciellt vid granskning av stora kodbaser. På liknande sätt argumenterar Sharma och Spinellis (2020) att traditionella mått som cyklomatisk komplexitet inte alltid speglar hur komplex koden faktiskt upplevs.

Mjuka tröskelvärden, så som de tillämpas i denna studie, gör i stället kontexten central för bedömningen. AI-reviewern instrueras att läsa den faktiska koden och avgöra om ett identifierat problem är skadligt i sitt sammanhang, snarare än att enbart reagera på att ett mätvärde överskridits. Detta möjliggör en mer nyanserad utvärdering där varje fil kan bedömas utifrån sitt eget syfte, sin design och sin roll i den övergripande systemkontexten. En klass som av designmässiga skäl hanterar flera orelaterade uppgifter får naturligt låga sammanhållningsvärden utan att detta utgör ett kvalitetsproblem. Med hårda tröskelvärden hade sådana klasser underkänts, trots att deras struktur är välmotiverad.

Ett ytterligare designval som syftar till att minska variationen i AI-reviewerns bedömningar och därmed öka determinismen är tvåstegs-prompten. Genom att tvinga modellen att först genomföra en strukturerad analys, och därefter fatta beslutet utifrån denna analys, styrs modellens resonemang in i ett mer förutsägbart mönster. Eftersom modellen följer samma analytiska struktur vid varje körning blir bedömningen mindre känslig för slumpmässiga variationer i hur modellen väljer att angripa problemet. Detta kan jämföras med en enstegsprompt där modellen själv avgör i vilken ordning den analyserar koden och fattar beslutet, vilket öppnar för större variation mellan körningar.

Genom att separera analys och beslut i två tydligt avgränsade steg, där det andra steget enbart bygger på det första steget, minskar utrymmet för att modellen ska resonera olika från gång till gång.

Detta innebär att utformningen av utvärderingsramverket bygger på en medveten avvägning mellan determinism och kontextkänslighet. Genom att kombinera deterministiska algoritmiska mått som input med mjuka, kontextkänsliga beslutsregler i AI-reviewern eftersträvas en bedömning som är så reproducerbar som möjligt utan att förlora den nyans som krävs för att undvika falska positiva resultat. Det är samtidigt viktigt att vara tydlig med att detta inte gör utvärderingen helt deterministisk. Så länge det slutgiltiga omdömet fattas av en LLM kommer en viss grad av variation att kvarstå mellan körningar. Frågan om hur denna variation påverkar utvärderingens tillförlitlighet är ett område som vore värdefullt att undersöka närmare i framtida arbete.

7.4 Gamla kodmått i en ny AI-driven kontext

En central aspekt som förtjänar att diskuteras är att flera av de algoritmiska kodutvärderingsmått som används i denna studie är relativt gamla. Som tidigare nämnt i avsnitt 3.2.1 utvecklade cyklomatisk komplexitet och Halstead-måtten under 1970-talet, och underhållbarhetsindex under 1990-talet. Trots sin ålder tillämpas måtten fortfarande i stor utsträckning i industrin, exempelvis använder Microsoft flera olika kvalitetsmått, bland annat cyklomatisk komplexitet, underhållbarhetsindex och sammanhållning (Microsoft, 2025f). Att måtten fortfarande används i branschen tyder på att de fortfarande säger något meningsfullt om koden, men det väcker samtidigt frågan om de verkligen är ändamålsenliga för dagens kodbaser. Sharma och Spinellis (2020) lyfter just denna problematik och menar att Halstead-volym och cyklomatisk komplexitet, som underhållbarhetsindex bygger på, se ekvation (1), utvecklades under en tid då mjukvarusystem var betydligt mindre och enklare. Därför misslyckas dessa mått att fånga centrala programegenskaper på arkitekturnivå i moderna kodbaser.

Denna problematik blir ännu mer angelägen i ljuset av den snabba utveckling som AI-driven mjukvaruutveckling genomgått. När AI-genererad kod blir allt vanligare förändras inte bara hur kod produceras, utan också hur den behöver granskas. Flera av utvecklarna på företaget beskriver att förhållandet mellan kodutveckling och kodgranskning har förändrats i grunden. Som en av utvecklarna uttryckte det har balansen skiftat från att man tidigare byggde kod i tio timmar och granskade i en, till att man idag genererar kod i en timme och granskar i tio. Dessutom beskriver utvecklarna i AI first-teamet att volymen kod som genereras snabbt blir så stor att manuell granskning av all kod blir orealistisk. Detta ställer i sin tur högre krav på de verktyg och mått som används för att bedöma kodkvalitet.

Här belyses eventuellt en forskningsbrist. Trots att kodgranskning har fått en allt viktigare roll i och med att de kodbaser som granskas i allt högre grad är AI-genererade, baseras de etablerade kodkvalitetsmåtten fortfarande på en kontext där koden skrevs manuellt och där kodbaserna var betydligt mindre. Eltabakh, Nabil Soudi och Shawky (2024) visar exempelvis att AI-genererad kod tenderar att uppvisa goda värden enligt traditionella mått, samtidigt som det är osäkert om dessa mått verkligen fångar kvalitet i en större systemkontext. Det utvärderingsramverk som tagits fram i denna studie, där algoritmiska mått kombineras med en AI-reviewer som gör kontextkänsliga bedömningar, kan ses som

ett sätt att hantera denna problematik. Genom att inte enbart förlita sig på de traditionella måtten utan att istället låta dem tolkas med bakgrund mot kodens kontext.

7.5 Utvärderingsramverkets syfte och funktion

Som ovan beskrivits upplever utvecklare att de idag lägger betydligt mer tid på kodgranskning än vad de gjort tidigare. Detta väcker på nytt frågan om vilken typ av kodgranskningsverktyg som eftersträvas och efterfrågas. Om det är i syfte att avlasta den mänskliga kodgranskningen och därmed bör ligga i linje med semantisk förståelse och upplevelsen av kodkvalitet, eller om det ska fungera som ett komplement med syfte att fånga kvalitetsaspekter som är svåra för mänskliga utvecklare att upptäcka. Exempelvis framhöll en av utvecklarna, i fallet med hög cyklomatisk komplexitet i valideringen, att hon hade föredragit om if-satser hade ersatts med switch-satser. Detta ligger i linje med det Sharma och Spinellis (2020) beskriver, att switch-satser upplevs som kognitivt mindre komplexa, trots att de bidrar lika mycket till den cyklomatiska komplexiteten som if-satser.

Om syftet är att skapa ett utvärderingsverktyg som efterliknar mänsklig granskning bör detta hanteras på ett sätt, exempelvis genom att inte flagga fall där switch-satser bidrar till hög cyklomatisk komplexitet. Om syftet däremot är att skapa ett komplement till mänsklig kodgranskning, är ett annat tillvägagångssätt mer lämpligt, exempelvis att fortsatt flagga hög cyklomatisk komplexitet oavsett om den uppstår genom if-satser eller switch-satser, eftersom verktygets funktion då är att synliggöra strukturella risker som mänskliga granskare kan uppleva som mindre problematiska.

Vidare berör detta toleransen för hur perfekt resultat utvärderingsramverket ska ge för att anses ”fungera”. Beroende på vilken typ av kodgranskning som eftersträvas, om den ska avlasta, komplettera eller till och med ersätta utvecklare, varierar också kraven på huruvida den ska leverera en bedömning som ligger i linje med kraven som ställs på mänskliga utvecklare, eller om den ska prestera bättre eller till och med felfritt.

I praktiken framstår kodgranskning sällan som något där perfektion är förväntad, utan flera utvecklare beskriver att det kan åsidosättas på grund av tidsbrist. En utvecklare hanterar som tidigare beskrivits det genom att dela upp i sådant som är ”must have” respektive ”nice to have”. Det antyder att även mänsklig kodgranskning präglas av prioriteringar och att alla potentiella problem inte nödvändigtvis fångas upp. Mot den bakgrunden blir frågan inte enbart om utvärderingsramverket kan leverera en perfekt och felfri bedömning utan snarare om det kan bidra med tillräckligt värde för att avlasta utvecklare i det tidskrävande och repetitiva granskningsarbetet. Särskilt vid granskning av AI-genererad kod, då kodvolymen kan bli mycket omfattande.

Detta ligger i linje med hur Batte (2025) beskriver kodutvärderingens funktion för att underlätta samarbete runt kod, samtidigt som han förklarar att det ofta åsidosätts på grund av tids- och resursbegränsningar. Han menar därför att AI-verktyg kan fylla en avlastande funktion genom att flagga för avvikelser, även om han anser att de inte fullt ut kan ersätta mänsklig kodgranskning.

Detta talar för att utvärderingsramverkets värde inte enbart bör bedömas utifrån hur väl det reproducerar mänsklig bedömning punkt för punkt, utan också utifrån vilken funktion det ska fylla i granskningsprocessen. Om målet är att avlasta utvecklare och frigöra tid för mer avancerade diskussioner om kvalitet, risk och strategi, kan utvärderingsramverket

anses fungera väl, även om inte alla aspekter som en erfaren utvecklare uppmärksammar fångas. Är målet däremot att ersätta mänsklig kodgranskning ökar kraven på precision, kontextförståelse och förmåga att hantera semantiska och designrelaterade uppgifter.

I litteraturstudien framkom att Scalabrino et al. (2021), vid tidpunkten för sin publikation, drog slutsatsen att det ännu inte fanns verktyg för att automatiskt mäta mänsklig kodförståelse. Författarna menade att begriplighet hos kod i högre grad beror på utvecklaren än enbart på den kod som granskas, vilket gör subjektiva aspekter svåra att fånga med traditionella algoritmiska mått. På liknande sätt framhåller även Oliveira et al. (2021) samt Börstler et al. (2023) att numeriska mått i sig inte ger tillräckligt underlag för att dra meningsfulla slutsatser om kodkvalitet, utan ytterligare tolkningsstöd krävs för att fånga aspekter såsom struktur, läsbarhet och förståelighet.

Samtidigt visar vår intervjustudie att utvecklarna återkommande lyfter aspekter kring vad som gör kod läsbar och lätt att förstå. Resultaten från valideringen indikerar också att flera av dessa aspekter fångas av utvärderingsramverket. Våra resultat tyder därmed på att AI-baserade granskningsverktyg kan bredda vad som är möjligt att fånga i en automatiserad kodutvärdering, jämfört med vad som tidigare har ansetts möjligt med enbart traditionella algoritmiska mått. Detta innebär inte att tidigare forskning motbevisas, men det antyder att dess slutsatser kan behöva nyanseras i ljuset av dagens AI-baserade verktyg. Författarnas resonemang om begränsningarna i traditionella algoritmiska mått framstår fortfarande som giltiga. Våra resultat tyder dock på att en kombination av traditionella, algoritmiska mått och en AI-reviewer som instruerats med en utvärderingsprocess som efterliknar mänskliga kodgranskningsstrategier, kan komma närmare en kodgranskning som på ett effektivt sätt tar mänsklig kodförståelse i beaktande.

8. Slutsats

Denna uppsats har undersökt hur kvaliteten hos AI-genererad kod kan utvärderas mot bakgrund av frågan om traditionella algoritmiska mått är tillräckliga, eller om kompletterande kodgranskningsmetoder krävs. Uppsatsen visar att traditionella algoritmiska kodutvärderingsmått, såsom cyklomatisk komplexitet, underhållbarhetsindex, sammanhållning och kodduplicering, fångar centrala, strukturella aspekter av kodkvalitet hos AI-genererad kod. De är effektiva för att identifiera mätbara strukturella egenskaper och avvikelser i förhållande till en kodbas norm. Däremot är de otillräckliga för att fånga mer subtila kvalitetsaspekter som utvecklare värdesätter, exempelvis semantiska val, designrelaterade beslut, kontextuell rimlighet och kodens estetiska uttryck. Vidare är flera av måtten utvecklade i en tid då kodbaser såg fundamentalt annorlunda ut än idag, vilket innebär att deras ändamålsenlighet i en AI-driven utvecklingskontext kan ifrågasättas.

De algoritmiska mått som i denna studie bedömts mest relevanta är cyklomatisk komplexitet både på fil- och metodnivå, underhållbarhetsindex, sammanhållningsmåtten TCC och LCOM1 samt kodduplicering. Dessa fångar tillsammans strukturella egenskaper hos enskilda filer och metoder respektive övergripande mönster i kodbasen. Utöver detta är kravuppfyllelse i förhållande till en specifikation ett centralt utvärderingskriterium, vilket också speglas i hur utvecklarna i intervjustudien beskrev sin egen kodgranskningsstrategi. Inget enskilt mått är dock tillräckligt i sig, utan ger störst analytiskt värde när de tolkas i kombination och i relation till kodens kontext.

Även om de algoritmiska kodutvärderingsmåtten har visat sig ge värde vid granskning av kod, framgår det av denna studie att de inte är tillräckliga som ensamt utvärderingsverktyg. Detta gäller oavsett om koden är AI-genererad eller skriven av mänskliga utvecklare, vilket går i linje med Oliveira et al. (2021) samt Börstler et al. (2023) som säger att vissa numeriska mått behöver kompletteras med ytterligare tolkningsstöd för att fånga aspekter som struktur, läsbarhet och förståelse. Måtten fångar visserligen centrala strukturella egenskaper, men enligt Oliveira et al. (2021) misslyckas de med att återspegla kodens specifika uppgift, syfte och roll i en större systemkontext. För att utvärderingen ska bli meningsfull behöver de algoritmiska måtten därför kompletteras med en kontextkänslig bedömning. I denna studie har detta uppnåtts genom att kombinera de algoritmiska måtten med en AI-reviewer som tolkar måtten i förhållande till kodens specifika uppgift, design och roll i systemkontexten. Vidare har följande designval gjorts, med utgångspunkt i tidigare forskning, i syfte att stärka utvärderingens tillförlitlighet: mjuka snarare än hårda tröskelvärden för att undvika falska positiva resultat och möjliggöra en kontextkänslig bedömning, jämförelsevärden från den befintliga kodbasen för att synliggöra avvikelser i förhållande till kodbasens normer, samt en tvåstegs-prompt för att stärka determinism och reproducerbarhet i AI-reviewerns bedömningar.

Vid utvärdering av specifikt AI-genererad kod blir behovet av ett automatiserat utvärderingsverktyg särskilt påtagligt. AI-genererade kodbaser tenderar att bli väldigt stora, vilket gör manuell mänsklig granskning av all kod orealistisk. Därtill har förhållandet mellan kodutveckling och kodgranskning skiftat i grunden i takt med att AI används för att generera kod, där utvecklare beskriver att granskning idag tar betydligt mer tid än själva utvecklingen. Detta innebär att behovet av tillförlitliga och kontextkänsliga utvärderingsverktyg som kan automatisera kodgranskningsprocessen ökar markant.

I förlängningen kan denna utveckling också innebära en förändrad syn på vad som är en rimlig granskningsnivå. I takt med att mängden genererad kod ökar blir det allt mindre realistiskt att förvänta sig att utvecklare ska orka eller hinna granska varje kodrad med samma noggrannhet som tidigare. Samtidigt ökar utvecklarnas förtroende för AI-genererad kod när LLM:erna blir allt mer kraftfulla, vilket i sin tur kan leda till en minskad benägenhet att granska och utvärdera koden. En sådan utveckling behöver inte innebära att kraven på kvalitet sänks, men däremot att ansvaret för kvalitetsbedömningen i högre grad förskjuts från enbart manuell granskning till en kombination av automatiserade verktyg och selektiv mänsklig bedömning. Det innebär att den manuella granskningen koncentreras till de delar av koden som bedöms vara mest kritiska eller riskfyllda, medan övriga delar i högre grad hanteras av automatiserade verktyg. I praktiken kan detta också innebära att vissa fel i AI-genererad kod behöver hanteras på liknande sätt som fel i mänskligt skriven kod, det vill säga som något som trots granskning ibland först upptäcks som buggar i produktion. Detta förstärker ytterligare behovet av automatiserade robusta utvärderingsverktyg som möjliggör en mer skalbar och tillförlitlig kvalitetsgranskning av stora AI-genererade kodbaser.

I ett vidare perspektiv väcker resultaten också frågor om hur framtidens granskningskompetens kommer att utvecklas. Studien tyder på att god mänsklig kodgranskning i hög grad bygger på erfarenhet, där utvecklare genom sin erfarenhet får en intuitiv förståelse för funktionalitet, buggrisker och rimlighet i kod. Om en allt större del av kodproduktionen sker genom hjälp av generativ AI kan juniora utvecklare komma att få färre tillfällen att själva skriva, ombearbeta och granska kod som bygger upp den

erfarenhetsbaserade kompetens och intuitiva förståelse som dagens seniora utvecklare ofta besitter. Detta kan i sin tur öka behovet av automatiserade och tillförlitliga utvärderingsverktyg, både eftersom volymen av genererad kod ökar och eftersom den mänskliga granskningskompetensen kan komma att försämras över tid. I ett sådant framtidsscenario blir frågan om hur kodkvalitet ska utvärderas inte bara en teknisk fråga, utan också en fråga om hur kunskap, ansvar och kvalitetssäkring ska organiseras i framtidens mjukvaruutveckling.

Referenser

Agarwal, Shivam (2025). Unit Test Best Practices: Top 10 Tips with Examples https://dualite.dev/blogs/unit-test-best-practices [Hämtad: 2026-04-08].

Artifical Analysis (2026). MiniMax-M2.7 vs Claude Opus 4.6 (Adaptive Reasoning, Max Effort): Model Comparison. https://artificialanalysis.ai/models/comparisons/minimax-m2-7-vs-claude-opus-4-6-adaptive [Hämtad: 2026-05-20].

Batte, Benjamin (2025). The Evolving Landscape of Code Review: Leveraging Artificial Intelligence for Enhanced Software Quality and Developer Productivity. https://www.ssrn.com/abstract=5214508 [Hämtad: 2026-02-04].

Bieman, James M. och Kang, Byung-Kyoo (1995). Cohesion and reuse in an object-oriented system. *ACM SIGSOFT Software Engineering Notes*, 20(SI), s. 259–262.

Busetto, Loraine, Wick, Wolfgang och Gumbinger, Christoph (2020). How to use and assess qualitative research methods. Neurological Research and Practice, 2(1), Artikel 14, doi:10.1186/s42466-020-00059-z.

Börstler, Jürgen, Bennin, Kwabena E., Hooshangi, Sara, Jeuring, Johan, Keuning, Hieke, Kleiner, Carsten, MacKellar, Bonnie, Duran, Rodrigo, Störrle, Harald, Toll, Daniel och Van Assema, Jelle (2023). Developers talking about code quality. *Empirical Software Engineering*, 28(6), s. 128, doi:10.1007/s10664-023-10381-0.

Chidamber, S. R. och Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), s. 476–493.

Ciceri, Christian, Farley, Dave, Ford, Neal, Harmel-Law, Andrew, Keeling, Michael, Lilienthal, Carola, Rosa, João, Zitzewitz, Alexander von, Weiss, René och Woods, Eoin (2022). *Software architecture metrics: case studies to improve the quality of your architecture*. First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly.

Decerno (2026). *Om Decerno – IT-konsulter med digitalt hantverk sedan 1984.* https://www.decerno.se/om-oss/ [Hämtad: 2026-06-16].

Eltabakh, Tasneem Muhammed, Nabil Soudi, Nada och Shawky, Doaa (2024). Quality of AI-Generated vs. Human-Generated Code. In: *2024 34th International Conference on Computer Theory and Applications (ICCTA)*. Alexandria, Egypt: IEEE, s. 200–205.

Eriksson, Lars Torsten, Wiedersheim-Paul, Finn (2011). *Att utreda, forska och rapportera.* 9 upplagan. Stockholm: Liber AB.

Ford, Neal, Richards, Mark, Sadalage, Pramod J. och Dehghani, Zhamak (2021). *Software architecture: the hard parts: modern trade-off analyses for distributed architectures*. First edition. Cambridge, Massachusetts London, England: The MIT Press.

ISO/IEC (2016) Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality. Tech. Rep. ISO/IEC 25023:2016, International Organization for Standardization, Geneva, Switzerland

Maikantis, Theodoros, Natsiou, Iliana, Ampatzoglou, Apostolos, Chatzigeorgiou, Alexander, Xinogalos, Stelios och Mittas, Nikolaos (2024). What you See is What you Get: Exploring the Relation between Code Aesthetics and Code Quality. In: *Proceedings of the 7th ACM/IEEE International Conference on Technical Debt*. Lisbon Portugal: ACM, s. 1–10.

Martin, Robert C., Martin, Micah D. och Martin, Justin M. (2026). *Clean code: a handbook of agile software craftsmanship*. Second edition. Hoboken, New Jersey: Addison-Wesley.

Microsoft (2026). Entity Framework. https://learn.microsoft.com/enus/dotnet/csharp/fundamentals/types/classes [Hämtad: 2026-03-17].

Microsoft (2025a). Code metrics – Maintainability index range and meaning.

https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=visualstudio [Hämtad: 2026-05-18].

Microsoft (2025b). Explore object oriented programming with classes and objects.

https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/classes [Hämtad: 2026-04-10].

Microsoft (2025c). Static Classes and Static Class Members (C# Programmning Guide). https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members [Hämtad: 2026-03-17].

Microsoft (2025d). Properties (C# Programming Guide) https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties [Hämtad: 2026-03-17].

Microsoft (2025e). Indexers. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/?source=recommendations [Hämtad: 2026-03-17].

Microsoft (2025f). Code metrics values.

https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=visualstudio [Hämtad: 2026-05-20].

Microsoft (2023a). Member Overloading. https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/member-overloading [2026-03-17].

Microsoft (2023b). Fields (C# Programming Guide). https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/fields [2026-03-17].

Microsoft (2023c). CA1502: Avoid excessive complexity. https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca1502 [2026-05-18].

Microsoft (2022). Inheritance – derive types to create more specialized behavior. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/inheritance [2026-03-17].

Microsoft (2021). Abstract and Sealed Classes and Class Members (C# Programming Guide) https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members [2026-04-10].

Nelson, Stacy och Schumann, Johann (2004). What makes a code review trustworthy? In: *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*. s. 1–10. doi:10.1109/HICSS.2004.1265711.

Oliveira, Joao C. B., Rios, Ricardo A., De Almeida, Eduardo S., Sant'Anna, Claudio N. och Rios, Tatiane Nogueira (2021). Fuzzy Software Analyzer (FSA): A New Approach for Interpreting Source Code Versioning Repositories. I: *2021 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. Luxembourg, Luxembourg: IEEE, s. 1–6.

Rivest, R. (1992). *The MD5 Message-Digest Algorithm*. RFC Editor. No. RFC1321. https://www.rfc-editor.org/info/rfc1321

Rosenberg, L., Hammer, T., Shaw, J. (1998). Software Metrics and Reliability (Proceedings of IEEE International Symposium on Software Reliability Engineering)

Scalabrino, Simone, Bavota, Gabriele, Vendome, Christopher, Linares-Vásquez, Mario, Poshyvanyk, Denys och Oliveto, Rocco (2021). Automatically Assessing Code Understandability. *IEEE Transactions on Software Engineering*, 47(3), s. 595–613.

Sharma, Tushar och Spinellis, Diomidis (2020). *Do We Need Improved Code Quality Metrics?* arXiv.org. https://arxiv.org/abs/2012.12324v1 [Hämtad: 2026-05-21].

Simons, Eric (2017). Introducing RealWorld. *Medium*, 2017-04-28. https://medium.com/@ericsimons/introducing-realworld-6016654d36b5 [Hämtad: 2026-05-18].

Tan, Leanne (2026). Yes, you're absolutely right… Right?: A mini survey on LLM sycophancy. *AI Practice and Data Engineering Practice, GovTech*, 2026-01-30. https://medium.com/dsaid-govtech/yes-youre-absolutely-right-right-a-mini-survey-on-llm-sycophancy-02a9a8b538cf [Hämtad: 2026-05-29].

Thomas, David och Hunt, Andrew (2020). *The pragmatic programmer: your journey to mastery*. Second edition, 20th anniversary edtion. Boston: Addison-Wesley

Tonella, Paolo och Abebe, Surafel Lemma (2009). Code quality from the programmer's perspective. In: *Proceedings of XII Advanced Computing and Analysis Techniques in Physics Research — PoS(ACAT08)*. Erice, Italy: Sissa Medialab, s. 001.

Tong, Weixi och Zhang, Tianyi (2024). CodeJudge: Evaluating Code Generation with Large Language Models. In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Miami, Florida, USA: Association for Computational Linguistics, s. 20032–20051.

Tosi, Davide (2024). Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation. *Future Internet*, 16(6), s. 188, doi:10.3390/fi16060188.

Tree-sitter (u.å.). Introduction - Tree-sitter. https://tree-sitter.github.io/tree-sitter/ [Hämtad: 2026-03-17]

Wang, Xinchen, Hu, Ruida, Gao, Pengfei, Peng, Chao och Gao, Cuiyun (2025). An Agent-based Evaluation Framework for Complex Code Generation. I: *2025 40th*

IEEE/ACM International Conference on Automated Software Engineering (ASE). Seoul, Korea, Republic of: IEEE, s. 2427–2439.

Yetiştiren, Burak, Özsoy, Işık, Ayerdem, Miray och Tüzün, Eray (2023). Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. https://arxiv.org/abs/2304.10778 [Hämtad: 2026-02-05].

A Intervjuguide

Gå ett varv där alla får besvara:

- Vad är din roll i projektet?
- Hur länge de har jobbat i respektive roll?

Berätta bakgrund till AI first-projektet:

- Varför startade det?
- På vems begäran startade det?
- Hur länge har det pågått och hur länge förväntas det pågå?

Hur ser arbetsmetodiken ut:

- Vad betyder AI first i praktiken?
- I vilka delar av utvecklingsprocessen har AI varit mest respektive minst värdeskapande?
- Finns det situationer där ni aktivt väljer bort AI och i så fall vilka?
- Eftersträvas också automatisering eller är det effektivisering eller något helt annat?
- Hur jobbar ni med kodgranskning/PR, är det ni som utvecklare som godkänner eller baseras det på till exempel en AI-reviewer?

Kvalitet

- Har ni definierat vad kvalitet är, och i så fall, vad innebär det för er i teamet?
- Hur mäter/säkerställer ni kvalitet? Vilka mått kollar ni på? Använder ni en AI-reviewer?
- Upplever ni att det finns tillräckliga mått som fångar dessa aspekter?
- Hur säkerställer ni att det är rätt funktionalitet som genereras?
- Vad eftersträvar ni, har ni ett uttalat mål?
- En kodgranskning/mjukvaruutveckling som är jämlik med en människa eller ska den vara bättre eller till och med felfri?
- Hur säkerställer ni kvalitet i alla led, i de olika rollerna?
- Hur säkerställer ni att det ni utvecklar är kompatibelt med de era kollegor utvecklar?
- Hur jobbar ni med kvalitet i teamet?

Resultat

- Hur går mjukvaruutvecklingen?
- Vad är de största skillnaderna mot era tidigare uppdrag med mer traditionella metoder?
- Håller mjukvaran ni utvecklar den kvaliteten ni har som mål?
- Finns det några vanliga brister som ni ser vid generering av kod? Lyckas era kvalitetsgranskningar fånga upp detta?

Avsluta med att gå varvet runt:

- Vad har varit mest utmanande eller överraskande i projektet för dig personligen?

- Hur ser ni på AI firsts möjlighet att bli standard inom mjukvaruutveckling?

B Valideringsunderlag

Pull Request 1 – RealWorld

Taskbeskrivning

Create reusable authorization logic that verifies the current authenticated user is the author of an article before allowing edit or delete operations.

Technical Notes

SERVICE: Article Ownership Verification

PURPOSE: Verify that the current authenticated user is the author of a given article, used as a guard before edit and delete operations.

BUSINESS RULES:

- AR-ARTICLE-002: Only article author can edit
- AR-ARTICLE-003: Only article author can delete
- BR-ARTICLE-003: Author is immutable after creation

INPUT:

- article: The article entity (with AuthorId or Author navigation property)
- currentUsername: The authenticated user's username from JWT

PROCESS:

1) Get the article's author username
2) Compare with current authenticated username
3) If mismatch, deny the operation

OUTPUT:

- Allow operation if match
- Return 403 Forbidden if current user is not the article author

ERROR RESPONSE:

- 403 with appropriate error message when user is not the article owner

USAGE:

- Called by edit article endpoint (003-002)
- Called by delete article endpoint (003-003)

Kod

ArticleAuthorizationService.cs

```
using System.Threading;
using System.Threading.Tasks;
using Conduit.Application.Interfaces;
using Conduit.Domain.Entities;
using Conduit.Domain.Exceptions;
using Conduit.Infrastructure.Data;
using Microsoft.EntityFrameworkCore;

namespace Conduit.Infrastructure.Services;

public class ArticleAuthorizationService : IArticleAuthorizationService
{
    private readonly ConduitDbContext _dbContext;
    private readonly ICurrentUserAccessor _currentUserAccessor;

    public ArticleAuthorizationService(
        ConduitDbContext dbContext,
        ICurrentUserAccessor currentUserAccessor)
    {
        _dbContext = dbContext;
        _currentUserAccessor = currentUserAccessor;
    }

    public async Task<bool> IsArticleOwnerAsync(Article article, CancellationToken ca
ncellationToken = default)
    {
        var username = _currentUserAccessor.GetCurrentUsername();
        if (string.IsNullOrEmpty(username))
        {
            return false;
        }

        var author = await _dbContext.Persons
            .AsNoTracking()
            .FirstOrDefaultAsync(p => p.PersonId == article.AuthorId, cancellationTok
en);

        return author?.Username == username;
    }

    public async Task AssertArticleOwnershipAsync(Article article, CancellationToken
cancellationToken = default)
    {
        var username = _currentUserAccessor.GetCurrentUsername();
        if (string.IsNullOrEmpty(username))
        {
            throw new UnauthorizedException("User is not authenticated.");
        }

        var author = await _dbContext.Persons
            .AsNoTracking()
            .FirstOrDefaultAsync(p => p.PersonId == article.AuthorId, cancellationTok
en);
```

```
        if (author?.Username != username)
        {
            throw new ForbiddenException("You do not own this article.");
        }
    }
}
```

Pull Request 2 – Maya

Task Description

Admins create, edit and deactivate organisations (Avdelning departments and Grupp groups). The Group type is required for the group-project feature; the Avdelning type is required for time-reporting membership and Veckomeddelande.

Technical Notes

- Entities: Organisation (Id PK, OrganisationName varchar(100), OrganisationType enum {Avdelning=1, Grupp=2}, Selectable bool default true, StartDate date, ExpireDate date null).
- API: GET /api/organisations?type=&activeOn=; POST /api/organisations (201); PUT /api/organisations/{id} (rejects type change with 422); DELETE /api/organisations/{id} (422 if referenced).
- UI: Page 'Organisations (Organisationer)' at /admin/organisations — table Name | Type | StartDate | ExpireDate | Selectable | Actions. Toolbar Add + filter chips (All | Avdelning | Grupp | Active only).
- Modal form: Name (required, max 100), Type (radio Avdelning|Grupp; disabled in edit mode), StartDate (required), ExpireDate (optional, must be ≥ StartDate). Helper text on Grupp: 'Required for the Group Project Management feature; Veckomeddelande not available'.
- Helper text on Avdelning: 'Required for time reporting membership and Veckomeddelande (DR-ORG-009)'. Empty state 'No organisations'. Nav: group 'Admin', label 'Organisations (Organisationer)', path '/admin/organisations', visibility 'authenticated, role Admin'.
- Authorization: authenticated, role 'Admin' (write); GET allowed for 'authenticated' (used by org pickers everywhere).
- Business rules: VR-ORG-001 (required fields); VR-ORG-002 (Selectable=false hides from lists); DR-ORG-009 (Veckomeddelande only for Avdelning). Implementation order: 1. migration → 2. entity + enum + config → 3. handlers + validators (immutable type, reference check on delete) → 4. endpoint → 5. NSwag → 6. page → 7. nav. PROJECT_CONTEXT.md: Entities [Organisation → Organisations], Endpoints [GET+POST+PUT+DELETE /api/organisations → OrganisationsEndpoints], Routes [/admin/organisations → OrganisationsPage.tsx].

Acceptance Criteria

- Admin can create an organisation with name, type (Avdelning|Grupp), StartDate and optional ExpireDate
- OrganisationType cannot be changed after creation (returns 422)
- Setting Selectable=false hides the org from selection lists used by other features
- GET /api/organisations supports filtering by type and active-on-date
- Org with active members or referenced by orders/projects cannot be hard-deleted (422)

Kod

OrganisationModal.tsx

```
import { useState, useEffect, type FormEvent, type RefObject } from 'react';
import { X } from 'lucide-react';
import { OrganisationsClient, CreateOrganisationCommand, UpdateOrganisationCommand, O
rganisationDto } from '../../../web-api-client';
import { apiHttp } from '../../../api-http';
import { getFormError } from '../../../api-errors';
import { Input } from '../../../components/Input';
import { Button } from '../../../components/Button';
import { Label } from '../../../components/Label';

const client = new OrganisationsClient(undefined, apiHttp);

interface OrganisationModalProps {
  dialogRef: RefObject<HTMLDialogElement | null>;
  organisation: { id: number; organisationName: string; organisationType: number; sel
ectable: boolean; startDate: Date; expireDate: Date | null } | null;
  isCreating: boolean;
  onCreated: (dto: OrganisationDto) => void;
  onUpdated: (dto: OrganisationDto) => void;
  onClose: () => void;
}

export function OrganisationModal({ dialogRef, organisation, isCreating, onCreated, o
nUpdated, onClose }: OrganisationModalProps) {
  const [name, setName] = useState('');
  const [orgType, setOrgType] = useState<number>(1);
  const [selectable, setSelectable] = useState(true);
  const [startDate, setStartDate] = useState('');
  const [expireDate, setExpireDate] = useState('');
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [serverError, setServerError] = useState<string | null>(null);

  useEffect(() => {
    if (organisation) {
      setName(organisation.organisationName);
      setOrgType(organisation.organisationType);
      setSelectable(organisation.selectable);
      setStartDate(organisation.startDate.toISOString().split('T')[0]);
      setExpireDate(organisation.expireDate ? organisation.expireDate.toISOString().s
plit('T')[0] : '');
    } else {
      setName('');
      setOrgType(1);
      setSelectable(true);
      setStartDate(new Date().toISOString().split('T')[0]);
      setExpireDate('');
    }
    setServerError(null);
  }, [organisation]);

  const reset = () => {
    setName('');
    setOrgType(1);
    setSelectable(true);
```

```
    setStartDate('');
    setExpireDate('');
    setServerError(null);
    setIsSubmitting(false);
  };

  const close = () => {
    reset();
    onClose();
  };

  const handleSubmit = async (e: FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    if (!name.trim() || !startDate) return;
    setIsSubmitting(true);
    try {
      if (isCreating) {
        const dto = await client.organisationsPOST(new CreateOrganisationCommand({
          organisationName: name.trim(),
          organisationType: orgType,
          selectable,
          startDate: new Date(startDate),
          expireDate: expireDate ? new Date(expireDate) : undefined,
        }));
        onCreated(dto);
      } else if (organisation) {
        await client.organisationsPUT(organisation.id, new UpdateOrganisationCommand(
{
          id: organisation.id,
          organisationName: name.trim(),
          organisationType: orgType,
          selectable,
          startDate: new Date(startDate),
          expireDate: expireDate ? new Date(expireDate) : undefined,
        }));
        const dto = new OrganisationDto({
          id: organisation.id,
          organisationName: name.trim(),
          organisationType: orgType,
          selectable,
          startDate: new Date(startDate),
          expireDate: expireDate ? new Date(expireDate) : undefined,
        });
        onUpdated(dto);
      }
      close();
    } catch (err) {
      const error = getFormError(err, 'Failed to save organisation. Please try again.
');
      if (error) setServerError(error);
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <dialog ref={dialogRef} onClose={close} aria-labelledby="organisation-modal-
heading">
```

```
      <div className="bg-white rounded-xl border border-gray-200 shadow-xl p-6">
        <div className="flex items-center justify-between mb-4">
          <h3 id="organisation-modal-heading" className="text-base font-
semibold text-gray-900">
            {isCreating ? 'New organisation' : 'Edit organisation'}
          </h3>
          <button type="button" onClick={close} aria-label="Close" className="text-
gray-400 hover:text-gray-900 transition-colors">
            <X size={16} />
          </button>
        </div>
        <form onSubmit={handleSubmit} className="space-y-4">
          <div>
            <Label htmlFor="org-name">Name</Label>
            <Input
              id="org-name"
              type="text"
              autoFocus
              maxLength={100}
              value={name}
              onChange={e => { setServerError(null); setName(e.target.value); }}
            />
          </div>

          <div>
            <Label>Type</Label>
            <div className="flex gap-4 mt-1">
              <label className="flex items-center gap-2 cursor-pointer">
                <input
                  type="radio"
                  name="org-type"
                  checked={orgType === 1}
                  onChange={() => setOrgType(1)}
                  disabled={!isCreating}
                  className="w-4 h-4 border-gray-300 text-violet-600 cursor-pointer"
                />
                <span className="text-sm text-gray-900">Avdelning</span>
              </label>
              <label className="flex items-center gap-2 cursor-pointer">
                <input
                  type="radio"
                  name="org-type"
                  checked={orgType === 2}
                  onChange={() => setOrgType(2)}
                  disabled={!isCreating}
                  className="w-4 h-4 border-gray-300 text-violet-600 cursor-pointer"
                />
                <span className="text-sm text-gray-900">Grupp</span>
              </label>
            </div>
            {!isCreating && <p className="text-xs text-gray-500 mt-
1">Type cannot be changed after creation</p>}
            {isCreating && orgType === 2 && (
              <p className="text-xs text-gray-500 mt-
1">Required for the Group Project Management feature; Veckomeddelande not available</
p>
            )}
            {isCreating && orgType === 1 && (
```

```
              <p className="text-xs text-gray-500 mt-
1">Required for time reporting membership and Veckomeddelande</p>
            )}
          </div>

          <div className="flex items-start gap-3">
            <input
              id="org-selectable"
              type="checkbox"
              checked={selectable}
              onChange={e => setSelectable(e.target.checked)}
              className="mt-1 h-4 w-4 rounded border-gray-300 text-violet-600 cursor-
pointer"
            />
            <div>
              <Label htmlFor="org-selectable" className="cursor-pointer mb-0">
                Selectable
              </Label>
              <p className="text-xs text-gray-500 mt-0.5">
                When disabled, this organisation is hidden from selection lists
              </p>
            </div>
          </div>

          <div>
            <Label htmlFor="org-start-date">Start Date</Label>
            <Input
              id="org-start-date"
              type="date"
              value={startDate}
              onChange={e => { setServerError(null); setStartDate(e.target.value); }}
            />
          </div>

          <div>
            <Label htmlFor="org-expire-date">Expire Date (optional)</Label>
            <Input
              id="org-expire-date"
              type="date"
              value={expireDate}
              onChange={e => { setServerError(null); setExpireDate(e.target.value); }
}
            />
            {expireDate && startDate && new Date(expireDate) < new Date(startDate) &&
 (
              <p className="text-xs text-red-500 mt-
1">Expire date must be on or after start date</p>
            )}
          </div>

          {serverError && (
            <p role="alert" className="text-sm text-red-600">{serverError}</p>
          )}
          <div className="flex justify-end gap-2">
            <Button type="button" variant="secondary" onClick={close}>Cancel</Button>
            <Button type="submit" disabled={isSubmitting}>
              {isSubmitting ? (isCreating ? 'Creating…' : 'Saving…') : (isCreating ?
'Create' : 'Save')}
```

```
            </Button>
          </div>
        </form>
      </div>
    </dialog>
  );
}
```

OrganisationsPage.tsx

```
import { useState, useEffect, useRef } from 'react';
import { OrganisationsClient, OrganisationDto } from '../../../web-api-client';
import { apiHttp } from '../../../api-http';
import { Button } from '../../../components/Button';
import { Spinner } from '../../../components/Spinner';
import { OrganisationModal } from './OrganisationModal';

const client = new OrganisationsClient(undefined, apiHttp);

interface Organisation {
  id: number;
  organisationName: string;
  organisationType: number;
  selectable: boolean;
  startDate: Date;
  expireDate: Date | null;
}

const OrgTypeLabels: Record<number, string> = {
  1: 'Avdelning',
  2: 'Grupp'
};

export function OrganisationsPage() {
  const [organisations, setOrganisations] = useState<Organisation[]>([]);
  const [isLoading, setIsLoading] = useState(true);
  const [isError, setIsError] = useState(false);
  const [loadKey, setLoadKey] = useState(0);
  const [editingOrg, setEditingOrg] = useState<Organisation | null>(null);
  const [isCreating, setIsCreating] = useState(false);
  const modalRef = useRef<HTMLDialogElement>(null);
  const [filter, setFilter] = useState<'all' | 'avdelning' | 'grupp' | 'active'>('all
');

  useEffect(() => {
    let cancelled = false;
    setIsLoading(true);
    setIsError(false);
    client.organisationsAll(undefined, undefined)
      .then((orgs: OrganisationDto[]) => {
        if (!cancelled) {
          setOrganisations(orgs.map((o: OrganisationDto) => ({
            id: o.id!,
            organisationName: o.organisationName!,
            organisationType: o.organisationType!,
            selectable: o.selectable ?? true,
            startDate: new Date(o.startDate!),
            expireDate: o.expireDate ? new Date(o.expireDate) : null,
          })));
          setIsLoading(false);
        }
      })
      .catch(() => {
        if (!cancelled) {
          setIsLoading(false);
          setIsError(true);
```

```
      }
    });
  return () => { cancelled = true; };
}, [loadKey]);

const handleCreated = (dto: OrganisationDto) => {
  setOrganisations(prev => [...prev, {
    id: dto.id!,
    organisationName: dto.organisationName!,
    organisationType: dto.organisationType!,
    selectable: dto.selectable ?? true,
    startDate: new Date(dto.startDate!),
    expireDate: dto.expireDate ? new Date(dto.expireDate) : null,
  }]);
  setIsCreating(false);
  setLoadKey(k => k + 1);
};

const handleUpdated = (dto: OrganisationDto) => {
  setOrganisations(prev => prev.map(o => o.id !== dto.id ? o : {
    id: dto.id!,
    organisationName: dto.organisationName!,
    organisationType: dto.organisationType!,
    selectable: dto.selectable ?? true,
    startDate: new Date(dto.startDate!),
    expireDate: dto.expireDate ? new Date(dto.expireDate) : null,
  }));
  setEditingOrg(null);
  setLoadKey(k => k + 1);
};

const openCreate = () => {
  setIsCreating(true);
  setEditingOrg(null);
  modalRef.current?.showModal();
};

const openEdit = (org: Organisation) => {
  setIsCreating(false);
  setEditingOrg(org);
  modalRef.current?.showModal();
};

const isActiveOrg = (org: Organisation) => {
  const now = new Date();
  return org.startDate <= now && (org.expireDate === null || org.expireDate >= now)
;
};

const filteredOrganisations = organisations.filter(org => {
  switch (filter) {
    case 'avdelning':
      return org.organisationType === 1;
    case 'grupp':
      return org.organisationType === 2;
    case 'active':
      return isActiveOrg(org);
    default:
```

```
        return true;
    }
  });

  if (isLoading) return (
    <div className="flex items-center gap-2 text-gray-500">
      <Spinner className="h-4 w-4" />
      <span>Loading…</span>
    </div>
  );

  if (isError) return (
    <div className="text-sm text-red-600">
      Failed to load organisations.{' '}
      <button onClick={() => setLoadKey(k => k + 1)} className="underline hover:no-
underline">
        Try again
      </button>
    </div>
  );

  return (
    <>
      <div className="mb-6 flex items-center justify-between">
        <h1 className="text-3xl font-bold text-gray-
900">Organisations (Organisationer)</h1>
        <Button onClick={openCreate}>Add</Button>
      </div>

      <div className="mb-4 flex gap-2">
        <button
          onClick={() => setFilter('all')}
          className={`px-3 py-1.5 text-sm rounded-md transition-
colors ${filter === 'all' ? 'bg-violet-100 text-violet-700' : 'bg-gray-100 text-gray-
600 hover:bg-gray-200'}`}
        >
          All
        </button>
        <button
          onClick={() => setFilter('avdelning')}
          className={`px-3 py-1.5 text-sm rounded-md transition-
colors ${filter === 'avdelning' ? 'bg-violet-100 text-violet-700' : 'bg-gray-
100 text-gray-600 hover:bg-gray-200'}`}
        >
          Avdelning
        </button>
        <button
          onClick={() => setFilter('grupp')}
          className={`px-3 py-1.5 text-sm rounded-md transition-
colors ${filter === 'grupp' ? 'bg-violet-100 text-violet-700' : 'bg-gray-100 text-
gray-600 hover:bg-gray-200'}`}
        >
          Grupp
        </button>
        <button
          onClick={() => setFilter('active')}
```

```
          className={`px-3 py-1.5 text-sm rounded-md transition-
colors ${filter === 'active' ? 'bg-violet-100 text-violet-700' : 'bg-gray-100 text-
gray-600 hover:bg-gray-200'}`}
        >
          Active only
        </button>
      </div>

      {filteredOrganisations.length === 0 ? (
        <p className="text-sm text-gray-500">No organisations.</p>
      ) : (
        <div className="border border-gray-200 rounded-lg overflow-hidden">
          <table className="w-full">
            <thead className="bg-gray-50 border-b border-gray-200">
              <tr>
                <th className="text-left text-sm font-medium text-gray-700 px-4 py-
3">Name</th>
                <th className="text-left text-sm font-medium text-gray-700 px-4 py-
3">Type</th>
                <th className="text-left text-sm font-medium text-gray-700 px-4 py-
3">StartDate</th>
                <th className="text-left text-sm font-medium text-gray-700 px-4 py-
3">ExpireDate</th>
                <th className="text-left text-sm font-medium text-gray-700 px-4 py-
3">Selectable</th>
                <th className="text-right text-sm font-medium text-gray-700 px-4 py-
3">Actions</th>
              </tr>
            </thead>
            <tbody>
              {filteredOrganisations.map(org => (
                <tr key={org.id} className="border-b border-gray-200 last:border-0">
                  <td className="px-4 py-3 text-sm text-gray-
900">{org.organisationName}</td>
                  <td className="px-4 py-3 text-sm text-gray-
900">{OrgTypeLabels[org.organisationType] || 'Unknown'}</td>
                  <td className="px-4 py-3 text-sm text-gray-
900">{org.startDate.toLocaleDateString()}</td>
                  <td className="px-4 py-3 text-sm text-gray-
900">{org.expireDate ? org.expireDate.toLocaleDateString() : '-'}</td>
                  <td className="px-4 py-3 text-sm text-gray-900">
                    {org.selectable ? 'Yes' : 'No'}
                  </td>
                  <td className="px-4 py-3 text-right">
                    <div className="flex items-center justify-end gap-3">
                      <button
                        onClick={() => openEdit(org)}
                        className="text-xs text-gray-500 hover:text-gray-
700 transition-colors"
                      >
                        Edit
                      </button>
                    </div>
                  </td>
                </tr>
              ))}
            </tbody>
          </table>
```

```
        </div>
      )}

      <OrganisationModal
        dialogRef={modalRef}
        organisation={isCreating ? null : editingOrg}
        isCreating={isCreating}
        onCreated={handleCreated}
        onUpdated={handleUpdated}
        onClose={() => { setIsCreating(false); setEditingOrg(null); }}
      />
    </>
  );
}
```

Pull Request 3 – ShipmentRouter

Taskbeskrivning

Implement a C# class that calculates shipping fees for packages based on destination region, weight, and a set of optional service flags, and produces shipping labels for outbound packages. The fee calculation must support multiple geographic regions, weight-based base rates, express vs. standard handling, fragile-item handling, and insurance. The class should also be capable of printing formatted shipping label information to the console for tracking and dispatch purposes.

Technical Notes

- Language: C#
- Create a class named `ShipmentRouter`
- Include a method `DetermineShippingFee(string region, double weight, bool isExpress, bool isFragile, bool isInsured)` that returns the calculated fee
- Include a method `GenerateShippingLabel(string trackingNumber, string region, double weight)` that prints formatted label information to console
- Use a shared field for the default carrier name that is referenced by the label-generating methods
- Use `Console.WriteLine` for all output

Acceptance Criteria

- The class compiles and runs without errors
- `DetermineShippingFee` correctly handles all combinations of regions, weight tiers, and service flags
- `GenerateShippingLabel` produces formatted output including tracking number, region, weight, and carrier
- All methods are accessible from outside the class

Kod

ShipmentRouter.cs

```
namespace Conduit.Infrastructure.Services;

public class ShipmentRouter
{
    private static readonly string DefaultCarrier = "FastShip Logistics";

    private static readonly Dictionary<string, decimal> RegionBaseRates = new()
    {
        { "North", 5.00m },
        { "South", 5.50m },
        { "East", 6.00m },
        { "West", 6.50m },
        { "Central", 4.50m }
    };

    private static readonly Dictionary<string, decimal> WeightTierRates = new()
    {
        { "Light", 1.00m },
```

```
        { "Medium", 2.50m },
        { "Heavy", 5.00m },
        { "ExtraHeavy", 8.00m }
    };

    private const decimal ExpressFee = 10.00m;
    private const decimal FragileFee = 5.00m;
    private const decimal InsuranceFee = 7.50m;

    public decimal DetermineShippingFee(string region, double weight, bool isExpress,
 bool isFragile, bool isInsured)
    {
        if (string.IsNullOrEmpty(region))
        {
            throw new ArgumentException("Region is required.", nameof(region));
        }

        if (weight < 0)
        {
            throw new ArgumentException("Weight cannot be negative.", nameof(weight))
;
        }

        decimal baseRate = RegionBaseRates.TryGetValue(region, out var rate) ? rate :
 10.00m;

        string weightTier = GetWeightTier(weight);
        decimal weightRate = WeightTierRates.TryGetValue(weightTier, out var wr) ? wr
 : 3.00m;

        decimal totalFee = baseRate + weightRate;

        if (isExpress)
        {
            totalFee += ExpressFee;
        }

        if (isFragile)
        {
            totalFee += FragileFee;
        }

        if (isInsured)
        {
            totalFee += InsuranceFee;
        }

        return totalFee;
    }

    public void GenerateShippingLabel(string trackingNumber, string region, double we
ight)
    {
        PrintLabel(trackingNumber, region, weight);
    }

    public void GenerateBackupShippingLabel(string trackingNumber, string region, dou
ble weight)
```

```
    {
        PrintLabel(trackingNumber, region, weight);
    }

    private void PrintLabel(string trackingNumber, string region, double weight)
    {
        Console.WriteLine("=== SHIPPING LABEL ===");
        Console.WriteLine($"Tracking Number: {trackingNumber}");
        Console.WriteLine($"Region: {region}");
        Console.WriteLine($"Weight: {weight:F2} kg");
        Console.WriteLine($"Carrier: {DefaultCarrier}");
        Console.WriteLine("======================");
    }

    private static string GetWeightTier(double weight)
    {
        return weight switch
        {
            <= 1.0 => "Light",
            <= 5.0 => "Medium",
            <= 20.0 => "Heavy",
            _ => "ExtraHeavy"
        };
    }
}
```

Pull Request 4 – ShipmentRouter

Task beskrivning

Implement a C# class that calculates shipping fees for packages based on destination region, weight, and a set of optional service flags, and produces shipping labels for outbound packages. The fee calculation must support multiple geographic regions, weight-based base rates, express vs. standard handling, fragile-item handling, and insurance. The class should also be capable of printing formatted shipping label information to the console for tracking and dispatch purposes.

Technical Notes

- Language: C#
- All conditional branches must be wrapped in curly braces, including single-statement branches
- Includes a method: DetermineShippingFee
- Signature: public double DetermineShippingFee(string region, double weight, bool isExpress, bool isFragile, bool isInsured)
- Returns the total shipping fee

Acceptance Criteria

- A C# class is implemented to calculate shipping fees and generate shipping labels.
- Shipping fees are calculated based on destination region, package weight, and selected service options.
- The calculation supports multiple regions, weight-based pricing, express delivery, fragile handling, and insurance fees.
- Invalid inputs are validated and handled appropriately.
- A formatted shipping label can be printed to the console with package and pricing details.
- The implementation is covered by unit tests and compiles successfully without errors

Kod

ShipmentRouter.cs

```
namespace Conduit.Domain.Entities;

public class ShipmentRouter
{
    private static readonly Dictionary<string, double> RegionBaseRates = new()
    {
        { "NorthAmerica", 10.0 },
        { "SouthAmerica", 15.0 },
        { "Europe", 12.0 },
        { "Asia", 14.0 },
        { "Africa", 18.0 },
        { "Oceania", 16.0 }
    };
```

```
    private static readonly Dictionary<string, double> RegionWeightMultipliers = new(
)
    {
        { "NorthAmerica", 1.0 },
        { "SouthAmerica", 1.2 },
        { "Europe", 1.1 },
        { "Asia", 1.3 },
        { "Africa", 1.5 },
        { "Oceania", 1.4 }
    };

    private const double WeightTier1Limit = 5.0;
    private const double WeightTier2Limit = 10.0;
    private const double WeightTier3Limit = 20.0;
    private const double WeightTier1Rate = 1.0;
    private const double WeightTier2Rate = 1.5;
    private const double WeightTier3Rate = 2.0;
    private const double WeightTier4Rate = 2.5;
    private const double ExpressBaseFee = 5.0;
    private const double ExpressMultiplier = 1.5;
    private const double FragileSurcharge = 5.0;
    private const double InsuranceSurcharge = 10.0;
    private const double OverweightSurcharge = 15.0;
    private const double MaxWeight = 100.0;

    public double DetermineShippingFee(string region, double weight, bool isExpress,
bool isFragile, bool isInsured)
    {
        if (string.IsNullOrWhiteSpace(region))
        {
            throw new ArgumentException("Region cannot be null or empty.", nameof(reg
ion));
        }

        if (weight < 0)
        {
            throw new ArgumentException("Weight cannot be negative.", nameof(weight))
;
        }

        string normalizedRegion = NormalizeRegion(region);

        if (!IsValidRegion(normalizedRegion))
        {
            throw new ArgumentException($"Invalid region: {region}. Supported regions
: NorthAmerica, SouthAmerica, Europe, Asia, Africa, Oceania", nameof(region));
        }

        double baseRate = GetRegionBaseRate(normalizedRegion);
        double weightMultiplier = GetRegionWeightMultiplier(normalizedRegion);

        double weightFee = CalculateWeightFee(weight, weightMultiplier);
        double regionFee = baseRate * weightMultiplier;
        double total = regionFee + weightFee;

        if (weight > MaxWeight)
        {
            total = total + OverweightSurcharge;
```

```
    }

    if (isExpress)
    {
        total = CalculateExpressFee(total);
    }

    if (isFragile)
    {
        total = ApplyFragileSurcharge(total);
    }

    if (isInsured)
    {
        total = ApplyInsuranceSurcharge(total);
    }

    if (isExpress && isFragile)
    {
        total = total * 0.95;
    }

    if (isExpress && isInsured)
    {
        total = total * 0.93;
    }

    if (isFragile && isInsured)
    {
        total = total * 0.92;
    }

    if (isExpress && isFragile && isInsured)
    {
        total = total * 0.90;
    }

    return Math.Round(total, 2);
}

private static string NormalizeRegion(string region)
{
    if (region.Length == 0)
    {
        return region;
    }
    return char.ToUpper(region[0]) + region.Substring(1).ToLower();
}

private static bool IsValidRegion(string normalizedRegion)
{
    return RegionBaseRates.ContainsKey(normalizedRegion);
}

private static double GetRegionBaseRate(string normalizedRegion)
{
    if (RegionBaseRates.TryGetValue(normalizedRegion, out double rate))
    {
```

```
            return rate;
        }
        return 10.0;
    }

    private static double GetRegionWeightMultiplier(string normalizedRegion)
    {
        if (RegionWeightMultipliers.TryGetValue(normalizedRegion, out double multipli
er))
        {
            return multiplier;
        }
        return 1.0;
    }

    private static double CalculateWeightFee(double weight, double weightMultiplier)
    {
        double fee = 0.0;

        if (weight <= WeightTier1Limit)
        {
            fee = weight * WeightTier1Rate;
        }
        else if (weight <= WeightTier2Limit)
        {
            fee = WeightTier1Limit * WeightTier1Rate;
            fee = fee + ((weight - WeightTier1Limit) * WeightTier2Rate);
        }
        else if (weight <= WeightTier3Limit)
        {
            fee = WeightTier1Limit * WeightTier1Rate;
            fee = fee + ((WeightTier2Limit - WeightTier1Limit) * WeightTier2Rate);
            fee = fee + ((weight - WeightTier2Limit) * WeightTier3Rate);
        }
        else
        {
            fee = WeightTier1Limit * WeightTier1Rate;
            fee = fee + ((WeightTier2Limit - WeightTier1Limit) * WeightTier2Rate);
            fee = fee + ((WeightTier3Limit - WeightTier2Limit) * WeightTier3Rate);
            fee = fee + ((weight - WeightTier3Limit) * WeightTier4Rate);
        }

        return fee * weightMultiplier;
    }

    private static double CalculateExpressFee(double total)
    {
        return total + ExpressBaseFee;
    }

    private static double ApplyFragileSurcharge(double total)
    {
        return total + FragileSurcharge;
    }

    private static double ApplyInsuranceSurcharge(double total)
    {
        return total + InsuranceSurcharge;
```

```
    }

    public void PrintShippingLabel(string region, double weight, bool isExpress, bool
 isFragile, bool isInsured, string trackingNumber)
    {
        if (string.IsNullOrWhiteSpace(trackingNumber))
        {
            throw new ArgumentException("Tracking number cannot be null or empty.", n
ameof(trackingNumber));
        }

        if (string.IsNullOrWhiteSpace(region))
        {
            throw new ArgumentException("Region cannot be null or empty.", nameof(reg
ion));
        }

        if (weight < 0)
        {
            throw new ArgumentException("Weight cannot be negative.", nameof(weight))
;
        }

        double fee = DetermineShippingFee(region, weight, isExpress, isFragile, isIns
ured);

        Console.WriteLine("==========================================");
        Console.WriteLine("           SHIPPING LABEL                 ");
        Console.WriteLine("==========================================");
        Console.WriteLine($"Tracking Number: {trackingNumber}");
        Console.WriteLine($"Destination: {region}");
        Console.WriteLine($"Weight: {weight} kg");
        Console.WriteLine($"Services:");

        if (isExpress)
        {
            Console.WriteLine("  - EXPRESS DELIVERY");
        }
        else
        {
            Console.WriteLine("  - STANDARD DELIVERY");
        }

        if (isFragile)
        {
            Console.WriteLine("  - FRAGILE");
        }

        if (isInsured)
        {
            Console.WriteLine("  - INSURED");
        }

        Console.WriteLine("------------------------------------------");
        Console.WriteLine($"Total Shipping Fee: ${fee:F2}");
        Console.WriteLine("==========================================");
    }
}
```