Bacheloroppgave

Sikker AI-akselerert utvikling i smidige arbeidsflyter: Et rammeverk for småteam

Secure AI-Accelerated Development in Agile Workflows: A Framework for Small Teams

Joakim Larssen
Bendik Hagen
Krister Eriksen

BAO304 Bacheloroppgave
Cybersikkerhet
Kristiania University College, Bergen

Akademisk veileder: Frederic Dorn, Kristiania
Ekstern veileder: Martin Bergo, CTO, ToSee AS

1. Innledning

1.1 Bakgrunn og motivasjon

Kunstig intelligens har de siste årene endret måten programvare utvikles på. Verktøy som GitHub Copilot, Claude Code og ChatGPT gjør det mulig å generere kode, foreslå løsninger og automatisere repetitive oppgaver i en hastighet som var utenkelig for få år siden. Kontrollerte eksperimenter har vist produktivitetsøkninger på mellom 26 og 55 prosent ved bruk av slike verktøy (Peng et al., 2023; Cui et al., 2025), og adopsjonen i bransjen har vært rask.

Samtidig viser forskningen at AI-generert kode kan inneholde alvorlige sikkerhetssårbarheter. Akademiske studier har funnet sårbarheter i rundt 40 prosent av Copilot-generert kode i sikkerhetssensitive scenarioer (Pearce et al., 2022), mens enkelte bransjekilder rapporterer høyere tall, blant annet 62 prosent, men med mindre transparent metodegrunnlag (Cloud Security Alliance, 2025). Problemet forsterkes av det Klemmer et al. (2024) beskriver som en «illusjon av korrekthet» – utviklere stoler på AI-generert kode uten tilstrekkelig verifikasjon.

For små utviklingsteam og oppstartsbedrifter er denne spenningen mellom produktivitet og sikkerhet spesielt kritisk. Slike team har begrensede ressurser, sjelden dedikerte sikkerhetsroller, og et høyt behov for rask iterasjon. Mesteparten av eksisterende forskning har undersøkt AI-verktøyenes effekt i store organisasjoner som Microsoft, Google og Accenture. Det mangler empirisk forskning på hvordan små team kan utnytte AI-akselerert utvikling på en måte som balanserer hastighet med sikkerhet, arkitektonisk kvalitet og pålitelighet.

Denne oppgaven springer ut av et praktisk samarbeid med ToSee AS, et lite norsk AI-startup, der forfatterne gjennom ett semester deltok som utviklere og brukte AI-verktøy som Claude Code, Opencode og Codex CLI i daglig arbeid.

1.2 Formål

Formålet med denne bacheloroppgaven er todelt. For det første ønsker vi å undersøke hvordan AI-akselerte utviklingsverktøy påvirker produktivitet, kodekvalitet og sikkerhetspraksis i et lite utviklingsteam som arbeider smidig. For det andre ønsker vi å utvikle et praktisk rammeverk med anbefalinger for hvordan små team kan integrere slike verktøy i sine arbeidsflyter uten å gå på kompromisser med sikkerhet, arkitektur eller pålitelighet.

Oppgaven kombinerer erfaringsbasert forskning fra daglig bruk av AI-verktøy i et reelt prosjekt med et kontrollert sikkerhetseksperiment som evaluerer AI-generert kode på tvers av flere modeller og prompttyper.

1.3 Problemstilling og delspørsmål

Oppgavens overordnede problemstilling er:

Hvordan kan små team i en bedrift integrere AI-akselerert utvikling i smidige arbeidsmetoder for å balansere sikkerhet, arkitektur og pålitelighet, samtidig som de fremmer innovasjon og parallelt samarbeid?

For å besvare dette er følgende delspørsmål formulert:

DS1: Hvordan kan AI-verktøy (f.eks. kodegenerering, automatisert testing) implementeres systematisk for å ivareta cybersikkerhet, robust arkitektur og systempålitelighet i små team?

DS2: Hvordan veier man avveininger mellom sikkerhet og smidighet i AI-drevne prosesser, og hvilke metoder støtter parallelt arbeid (f.eks. via versjonskontroll og kollaborative plattformer)?

DS3: Hvordan kan moderne AI-teknologier og arbeidsmetoder (f.eks. DevSecOps, agile med AI) øke innovasjon og produktutvikling i en cybersikkerhetskontekst?

1.4 Avgrensninger

Oppgaven avgrenses på følgende måter:

Studien er gjennomført i konteksten av ett enkelt samarbeidsprosjekt med ToSee AS og er dermed en enkeltstående case-studie. Funnene er ikke direkte generaliserbare til andre organisasjoner, men kan gi overførbar innsikt til lignende kontekster.

AI-verktøyene som undersøkes er begrenset til de som ble brukt i prosjektperioden, primært Claude Code, Codex CLI og Opencode samt modeller evaluert i sikkerhetseksperimentet

(Claude Opus 4.6, GPT-5.3-Codex, Gemini 3.1 Pro og Kimi K2.5). Oppgaven dekker ikke alle tilgjengelige AI-utviklingsverktøy og modeller på markedet.

Sikkerhetseksperimentet er avgrenset til statisk analyse (SAST) med Semgrep og manuell gjennomgang, ikke dynamisk testing eller penetrasjonstesting av kjørende systemer.

Oppgaven fokuserer på webutvikling med React, Bun.js og relaterte teknologier, og funnene er mest direkte relevante for denne teknologistakken.

Utviklingsperioden strekker seg over ett semester, noe som begrenser antall sprinter og dermed omfanget av erfaringsdata.

1.5 Begrepsavklaringer

Følgende begreper brukes gjennomgående i oppgaven:

AI-akselerert utvikling – Programvareutvikling der AI-verktøy som kodegenerering, automatisk testing og intelligente forslag brukes aktivt for å øke utviklingshastigheten.

LLM (Large Language Model) – Store språkmodeller trent på omfattende datasett, som GPT-5, Claude og Gemini, som er i stand til å generere og forstå naturlig språk og programkode.

Smidig utvikling (agile) – Iterativ utviklingsmetodikk basert på hyppige leveranser, tett samarbeid og kontinuerlig tilpasning, eksempelvis Scrum og Kanban.

DevSecOps – En tilnærming som integrerer sikkerhetspraksis i alle faser av utviklings- og driftslivssyklusen, fremfor å behandle sikkerhet som et separat steg.

SAST (Static Application Security Testing) – Automatisert analyse av kildekode for å identifisere potensielle sikkerhetssårbarheter uten at koden kjøres.

Sikkerhetsprompt – En spesialisert systemprompt som instruerer AI-modellen om å prioritere sikker kodepraksis, inkludert input-validering, parameteriserte spørringer og unngåelse av hardkodede hemmeligheter.

Teknisk gjeld – Oppsamlede snarveier og suboptimale løsninger i kodebasen som øker fremtidig vedlikeholdskostnad.

Text-to-speech (TTS) – Teknologi som konverterer tekst til syntetisk tale, kjernen i produktet som utvikles hos ToSee AS.

2. Litteraturstudie

Dette kapittelet gjennomgår eksisterende forskning og faglitteratur om AI-verktøy i programvareutvikling. Gjennomgangen dekker produktivitetseffekter, sikkerhetssårbarheter i AI-generert kode, arkitektoniske tilnærminger, avveiningen mellom sikkerhet og

utviklingstempo, samt foreslåtte tiltak fra litteraturen. Kapittelet identifiserer også et forskningsgap som danner grunnlaget for denne oppgavens problemstilling.

2.1 Søkestrategi

Litteratursøket ble gjennomført ved bruk av følgende akademiske databaser: IEEE Xplore (fokus på programvareutvikling og tekniske studier) og ACM Digital Library (omfattende dekning av datavitenskap og software engineering). Søkene ble utført med kombinasjoner av nøkkelbegrep (AI-assisted software development, GitHub Copilot, generative AI developer productivity, AI code generation), sekundære termer (software team productivity, AI tools integration, code quality AI, developer experience AI) og kontekstspesifikke nøkkelbegrep (startup software development, small team AI tools, AI security software development).

2.2 Produktivitet og AI-verktøy

Kunstig intelligens (KI) har de siste årene transformert måten programvare utvikles på. Med introduksjonen av verktøy som GitHub Copilot, ChatGPT og Amazon CodeWhisperer har utviklere fått tilgang til AI-assistenter som kan generere kode, foreslå løsninger og automatisere repetitive oppgaver (Yetiştiren et al., 2023). Et kontrollert eksperiment med 95 profesjonelle utviklere viste at gruppen som brukte GitHub Copilot, fullførte en avgrenset oppgave om lag 55 % raskere enn kontrollgruppen (Peng et al., 2023). En annen studie basert på tre felteksperimenter med nærmere 5 000 utviklere hos Microsoft og Accenture fant en 26 % økning i fullførte oppgaver (Cui et al., 2025).

Bruken av AI-kodeverktøy har økt betydelig, og slike verktøy har blitt en del av mange utvikleres daglige arbeidsflyt. Samtidig reiser AI-assistert utvikling viktige spørsmål knyttet til kodekvalitet, sikkerhet og teamdynamikk. Studier har vist at AI-generert kode kan inneholde sårbarheter som utviklere overser på grunn av en «illusjon av korrekthet» (Klemmer et al., 2024). For små team og oppstartsbedrifter, der ressursene er begrensede og behovet for rask iterasjon er høyt, blir balansen mellom produktivitet og sikkerhet særlig kritisk.

Små utviklingsteam kan dra nytte av AI-verktøy gjennom raskere kodegenerering, støtte til testskriving og automatisering av repetitive oppgaver. Dette kan frigjøre tid til arkitektur, brukervennlighet og videre produktutvikling. Samtidig forutsetter gevinstene at teamet har strukturerte prosesser for testing, kodegjennomgang og sikkerhetskontroll. AI-verktøy bør derfor forstås som produktivitetsforsterkere, ikke som erstatning for utviklerkompetanse eller kvalitetssikring.

Effektstørrelsene varierer også med kontekst og oppgavetype. Laboratorieforsøk som Peng et al. (2023) gir høy intern validitet, men resultater fra virkelige prosjekter er ofte mer varierte og avhenger av kodebase, oppgavetype og utviklernes erfaring. Gevinstene er særlig tydelige i avgrensede og repetitive oppgaver, mens mer komplekse flerfil-kontekster krever mer manuell

vurdering. I praksis betyr dette at småteam kan få stor nytte av AI-verktøy, men bare dersom bruken kombineres med tydelige rutiner for validering, sikkerhet og arkitektonisk kontroll.

2.3 Dokumenterte sårbarheter i AI-generert kode

2.3.1 Omfang og statistikk

Forskningen viser at sikkerhetsfeil i AI-generert kode forekommer i betydelig omfang. Schreiber og Tippe (2025) analyserte 7 703 GitHub-filer eksplisitt merket som AI-genererte, og identifiserte 4 241 CWE-instanser fordelt på 77 sårbarhetskategorier ved hjelp av CodeQL. Cloud Security Alliance (2025) rapporterer, med referanse til Endor Labs, at 62 % av AI-genererte kodeløsninger inneholder designfeil eller kjente sikkerhetssårbarheter. Dette tallet bør tolkes med forsiktighet, ettersom primærkilden og metodegrunnlaget ikke er tydelig dokumentert. Veracode (2025), basert på 80 kodeoppgaver testet på tvers av over 100 språkmodeller, fant at 45 % av AI-generert kode inneholdt sikkerhetsfeil.

2.3.2 Hallusinerte pakker og avhengigheter

Et særlig alvorlig problem er «package hallucination» hvor AI-modeller genererer kode som refererer til ikke-eksisterende programvarepakker. Angripere kan utnytte dette ved å opprette ondsinnede pakker med de hallusinerte navnene, noe som fører til «supply chain attacks» (Ji et al., 2024). Georgetown CSET-rapporten kategoriserer dette som en indirekte sikkerhetsrisiko der AI-modellene selv er sårbare for manipulasjon.

2.3.3 Injection-sårbarheter

SQL-injection, command injection og cross-site scripting (XSS) er blant de vanligste sårbarhetene i AI-generert kode. Pearce et al. (2022) fant at omtrent 40 % av Copilot-generert kode inneholdt sikkerhetssårbarheter i en studie av 89 sikkerhetssensitive scenarioer spesifikt designet rundt MITREs Top 25 CWE-er. Det er viktig å merke seg at dette tallet gjelder bevisst utsatte høyrisikoscenarioer, og at sårbarhetsraten i generell koding sannsynligvis er lavere. JFrog (2024) sin analyse av kodegenerativ AI bekrefter at injection-sårbarheter og manglende input-validering er gjennomgående problemer.

2.3.4 Usikre standardinnstillinger

AI-modeller har en tendens til å generere kode med usikre standardinnstillinger, inkludert hardkodede hemmeligheter og credentials, svak eller fraværende kryptografisk praksis, manglende autentiserings- og autorisasjonsmekanismer, samt buffer overflow og minnehåndteringsfeil (Schreiber & Tippe, 2025).

2.3.5 Unike risikoer ved AI-generert kode

Ji et al. (2024) identifiserer tre brede risikokategorier: (1) modeller som genererer usikker kode, (2) modeller som selv er sårbare for angrep og manipulasjon, og (3) konsekvenser for cybersikkerhet videre i kjeden som feedback-loops i trening av fremtidige AI-modeller. Et kritisk poeng er at AI-generert kode ofte er unik og ikke gjenbruker eksisterende open source-komponenter. Dette betyr at sårbarheter ikke kan oppdages via kjente CVE-databaser, noe som bryter med den tradisjonelle sikkerhetsmodellen der felles kodebaser som OpenSSL og Log4j muliggjorde koordinert respons på sårbarheter (RunSafe Security, 2025).

2.4 Robust arkitektur og arkitektonisk styring

For å unngå at AI-generert kode gradvis svekker systemarkitekturen, finnes det foreslåtte arkitekturmønstre i praksislitteraturen. Ett slikt mønster er «Skeleton Architecture», der mennesker definerer de stabile rammeverkene, det Farry (2026) kaller «Stable Skeleton» og lar AI fylle inn forretningslogikk i form av «Vertical Tissue». Basisklasser kan fange opp felles krav på tvers av systemet som logging, autentisering og feilhåndtering, slik at AI ikke kan omgå sikkerhetskontroller (Farry, 2026). Det bør understrekes at dette er ett mulig arkitekturgrep blant flere, foreslått i en praksisartikkel, og ikke en allment etablert standard. Slik strukturell styring kan likevel hjelpe team med å isolere AI-generert funksjonalitet mens sentrale retningslinjer beholdes sentralt.

Sammenfattet handler implementeringen om å kombinere AI-verktøy med etablerte ingeniørprinsipper. Små team bør satse på kontinuerlig testing og kvalitetssikring – gjerne med AI-drevne testgeneratorer kombinert med valideringsrammeverk som sikrer konsistens og sporbarhet (Ramachandran, 2025) – solid modulær kode og automatiserte sikkerhetstiltak. ThoughtWorks understreker at en kvalitetsbevisst kodebase er en forutsetning for å lykkes med AI-akselerert utvikling (ThoughtWorks, 2025). Ved å integrere AI-verktøy i CI/CD-pipeline kan teamene oppnå høyere utviklingstakt uten å gå på akkord med sikkerhet eller pålitelighet (Fauscette, 2024).

2.5 Sikkerhet vs. smidighet – parallell utvikling

AI-drevne prosesser tvinger frem avveininger mellom sikkerhet og tempo. Strenge sikkerhetsprosedyrer kan gjøre team tregere, mens for høy hastighet kan introdusere sårbarheter. Små team må balansere DevSecOps-prinsipper: sikkerhet integrert i utviklingsløpet slik at det ikke blir et eksternt stoppunkt. Dagens beste praksis er å «skifte venstre» – automatisere sikkerhetsskanning, kodegjennomgang og samsvarsjekk i pipeline ved hver commit (Kalva, 2024). På den måten minimerer man overraskelser sent i syklusen og opprettholder både hurtighet og sikkerhet.

Moderne samhandlingsverktøy legger til rette for effektivt parallelt arbeid. Ved å benytte Git-baserte arbeidsflyter som kombinerer beskyttede hovedgrener med kortvarige utviklingsbranches, kan utviklerteam operere simultant uten risiko for destruktive overlapp i

kodebasen. Pull requests og «feature flags» kan brukes for å isolere eksperimenter og håndtere risiko. Atlassian poengterer at riktig bruk av versjonskontroll isolerer endringer, muliggjør parallelt arbeid og reduserer risiko knyttet til sammenslåing (Atlassian, u.å.). Smidige møter (scrum, standups) og kontinuerlig integrasjon sikrer at ny kode tidlig testes sammen.

Balansen oppnås ved automatiske sikkerhetsporter i utviklingsløpet: AI-drevne statiske analyser eller container-sikkerhetssjekker kan blokkere pushes med kritiske feil. Slike DevSecOps-tiltak kan i praksis øke utviklingsfarten fordi de fjerner unødvendige manuelle godkjenninger sent i prosessen. For å forebygge at ansatte tyr til uoffisielle løsninger som medfører skjult sikkerhetsrisiko, bør organisasjoner tilrettelegge for smidige iterasjoner. Det er viktig å unngå at samtlige AI-verktøy begrenses av altfor streng kontroll, da praktisk erfaring tilsier at slike restriksjoner ofte fører til at sikkerhetsregler omgås. En undersøkelse gjennomført av Writer og Workplace Intelligence (2025) blant 1 600 respondenter viser at over en tredjedel (35 %) av ansatte betaler selv for AI-verktøy de bruker i jobben, fordi arbeidsgiveren ikke tilbyr ønskede løsninger. Selv om «betale selv» ikke nødvendigvis er det samme som «utenfor godkjente kanaler», peker funnet på et reelt risikobilde knyttet til bruk av uautoriserte verktøy og potensiell eksponering av sensitiv informasjon. Teamene bør derfor etablere trygge kanaler med godkjente verktøy og sikker autentisering, slik at sikkerhet og hurtighet kan sameksistere.

2.6 Foreslåtte tiltak fra litteraturen

2.6.1 Code review og manuell gjennomgang

Kashif et al. (2025) fant at 76,6 % av utviklere alltid eller noen ganger deklarerer AI-generert kode i sine prosjekter. Grunnene inkluderer behovet for å spore og overvåke koden for fremtidig gjennomgang og feilsøking. Tambon et al. (2025) understreker at AI-generert kode vanligvis krever gjennomgang før integrering i prosjekter. Anbefalt praksis er å behandle AI som en «junior utvikler» som krever code review for all generert kode.

2.6.2 SAST og automatisert sikkerhetstesting

Static Application Security Testing (SAST) er essensielt for å identifisere sårbarheter tidlig. Verktøy som CodeQL anbefales for statisk analyse av all kode, inkludert AI-generert (Schreiber & Tippe, 2025). Software Composition Analysis (SCA) bør brukes for å analysere avhengigheter, og secret scanning implementeres for å unngå lekkede credentials. Teamene bør innføre «shift-left»-prinsipper: innbygge sikkerhetstesting, statisk kodeanalyse og skanning av avhengigheter tidlig i CI/CD-pipelinen. Dette er i tråd med etablerte rammeverk fra blant annet OWASP og NIST (Scarfone et al., 2022) som eksplisitt anbefaler at sikre utviklingspraksiser integreres i hele utviklingssyklusen. AI-drevne kodeanalysatorer kan bidra til å identifisere sårbarheter mens koden skrives, og leverandørsiden (som GitLab) hevder at slike verktøy kan bidra til å redusere falske positiver – selv om dette ikke er dokumentert som en universell effekt (GitLab, u.å.; Kalva, 2024).

2.6.3 DevSecOps-praksis

DevSecOps integrerer sikkerhet i hver fase av DevOps-livssyklusen (Wiz, 2026). For små team anbefales shift-left security (integrer sikkerhetsverktøy i CI/CD-pipeline fra dag en), automatisering med open source-verktøy som GitLab CI/CD og Jenkins med sikkerhetsplugins, risikobasert prioritering (prioriter basert på exploitability og blast radius, ikke bare CVSS-score), samt security champions – utviklere med ekstra sikkerhetsansvar i teamet (Anchore, 2025). Bright Security (2025) understreker at i dagens raske, agile verden bør datasikkerhet være alles ansvar.

2.6.4 OWASP LLM Top 10

OWASP har utviklet en Top 10-liste spesifikt for Large Language Model Applications som er relevant for kodegenerering (OWASP, 2025). De mest relevante risikoene inkluderer LLM01: Prompt Injection (manipulering av AI-output gjennom ondsinnet input), LLM02: Insecure Output Handling (manglende validering av AI-output før bruk) og LLM03: Training Data Poisoning (forgifting av treningsdata som påvirker modellens output). Sonar (2025) gir praksisrettede anbefalinger for hvordan disse risikoene kan mitigeres i kontekst av kodegenerering.

2.7 DevSecOps, AI og økt innovasjon

Moderne metoder som DevSecOps og agile med AI legger grunnlaget for innovasjon selv i en sikkerhetskritisk kontekst. IBM konstaterer at DevSecOps' to sentrale fordeler er hurtighet og sikkerhet: ved å automatisere sikkerhetsgjennomgang sparer man tid og kostnader som ellers ville gått med til å utbedre problemer i etterkant (IBM, u.å.). Dette frigjør kapasitet til eksperimentering. ThoughtWorks (2025) påpeker at akselerasjon kan påvirke ikke bare selve utviklingen, men hele feedback-syklusen. Dersom teamet kan levere kode hyppigere og med samme kvalitet, kan produktstrategi og forretningsmuligheter endres ved at tid fra idé til produksjon kortes ned. Små utviklingsteam kan slik innovere kontinuerlig innenfor en forsvarlig sikkerhetsramme, der automatisert skanning av avhengigheter og smarte CI/CD-pipelines gjør at nye løsninger kan lanseres raskere uten å påføre organisasjonen unødig risiko (Taware, 2025; Kalva, 2024).

2.8 AI i smidig metodikk

Introduksjonen av AI-kodeverktøy påvirker smidige praksiser på flere nivåer. På individnivå viser studier at AI kan gi produktivitetsgevinster, særlig i avgrensede oppgaver og for utviklere med mindre erfaring (Peng et al., 2023; Cui et al., 2025). Samtidig viser nyere forskning at økt individuell produktivitet ikke automatisk gir bedre teamflyt, høyere leveranseflyt eller mer stabile leveranser.

På teamnivå er bildet mer sammensatt. DORA-rapporten for 2024, basert på om lag 39 000 respondenter, fant at økt AI-adopsjon var assosiert med noe høyere individuell produktivitet, men samtidig lavere leveransetakt og leveransestabilitet. Rapportens «vakuumhypotese» antyder at frigjort tid fra AI kan absorberes av lavere prioriterte oppgaver i stedet for å øke total leveransetakt. Dette er særlig relevant i smidige team, hvor produktivitet også avhenger av koordinering, prioritering og kvalitet.

METR-studien (Becker et al., 2025) nyanserer bildet ytterligere. I studien brukte erfarne open source-utviklere med AI-tilgang lengre tid enn utviklere uten AI, blant annet på grunn av gjennomgang, korrigering, overoptimisme og kontekstbytte. Dette antyder at AI gir størst effekt i tydelig avgrensede oppgaver, mens gevinsten kan reduseres i komplekse kodebaser der utvikleren allerede har høy kjennskap til kodebasen.

CodeRabbit (2025) fant i en analyse av 470 pull requests at AI-medforfattede bidrag hadde flere problemer enn menneskeskrevne bidrag, særlig knyttet til logikk, korrekthet og lesbarhet. Dette viser at smidige team må tilpasse review-prosessen når AI-generert kode inngår i arbeidsflyten. AI kan øke kodeproduksjonen, men samtidig øke behovet for strukturert gjennomgang og tydelige kriterier for godkjenning.

Utviklerundersøkelser viser også at tilliten til AI varierer mellom ulike faser av utviklingssyklusen. Stack Overflow Developer Survey 2025 viser skepsis til bruk av AI i deployment, monitorering og prosjektplanlegging, mens JetBrains State of Developer Ecosystem 2025 viser at AI-verktøy brukes regelmessig av mange utviklere, men at store tidsbesparelser ikke er universelle. Dette understreker at AI i smidig metodikk bør brukes selektivt: som støtte i oppgavenedbryting, koding, dokumentasjon og testing, men med større forsiktighet i overordnet prosjektplanlegging, sikkerhetskritiske vurderinger og produksjonsnære aktiviteter.

2.9 Teknisk gjeld i AI-akselerert utvikling

AI-akselerert utvikling introduserer nye former for teknisk gjeld som skiller seg fra tradisjonelle mønstre. GitClear (2025) analyserte 211 millioner endrede kodelinjer på tvers av tusenvis av repositorier over perioden 2020 til 2024. De fant at refaktorering falt fra 25 prosent til under 10 prosent av endrede linjer, mens kopiert kode økte fra 8,3 til 12,3 prosent. Dupliserte kodeblokker økte åtte ganger i 2024 sammenlignet med tidligere år. Rapporten konkluderer med at AI-generert kode ligner på bidrag fra midlertidige utviklere som er tilbøyelige til å bryte DRY-prinsippet (Don't Repeat Yourself).

I en bransjerapport analyserte OX Security (2025) over 300 open source-repositorier og identifiserte ti kritiske arkitektur- og sikkerhetsfallgruver i AI-generert kode. Blant funnene var kode som ignorerer veletablerte biblioteker til fordel for enklere implementasjoner, overfladisk testing med høye dekningsmål men lav faktisk kvalitet, og regresjon mot tettkoblede systemstrukturer. Rapporten beskriver det de kaller «Army of Juniors»-effekten: AI oppfører seg som talentfulle juniorutviklere som mangler strukturell dømmekraft og sikkerhetsbevissthet.

Et særskilt alvorlig problem er det Spracklen et al. (2025) dokumenterer i en studie akseptert for USENIX Security 2025. De testet 16 kodegenererende språkmodeller på 576 000 kodesampler og fant at i gjennomsnitt om lag 20 prosent av anbefalte pakker ikke eksisterte. Kommersielle modeller hadde en hallusinasjonsrate på 5,2 prosent, mens open source-modeller lå på 21,7 prosent. Til sammen identifiserte de 205 474 unike hallusinerte pakkenavn. Kritisk nok gjentok 58 prosent av de hallusinerte pakkene seg på tvers av kjøringer, og 43 prosent dukket opp hver gang. Dette gjør dem til forutsågbare mål for det som kalles «slopsquatting», der angripere registrerer ondsinnede pakker med de hallusinerte navnene på PyPI eller npm.

Moreschini et al. (2026) gjennomførte en bred litteraturgjennomgang publisert i Journal of Systems and Software som formelt kategoriserer nye former for teknisk gjeld fra AI-assistert utvikling. Gjennomgangen identifiserer gjeld knyttet til vedlikeholdbarhet, forklarbarhet og datastyring, og konkluderer med at håndtering av teknisk gjeld i AI-æraen krever kontinuerlige, automatiserte og tverrfaglige strategier.

2.10 Utfordringer for småteam

Små team og oppstartsbedrifter befinner seg i en paradoksal posisjon: de har potensielt mye å vinne på AI som produktivitetsforsterker, men er dårligere rustet til å håndtere risikoen. Georgetown CSET (Ji et al., 2024) påpeker at risikoene ved AI-kodegenerering ikke er jevnt fordelt, og at ressurssterke organisasjoner har en fordel.

Klemmer et al. (2024) fant i 27 utviklerintervjuer at AI-generert kode som standard utelater sikkerhetstiltak, som trygg passordhåndtering, med mindre utvikleren eksplisitt ber om det. Uten dedikert sikkerhetskompetanse går slike sårbarheter lett uoppdaget i småteam.

Verizon DBIR 2025 kvantifiserer trusselbildet: løsepengevirus var involvert i 88 prosent av datainnbrudd hos SMB, mot 39 prosent hos store virksomheter. Tre angrepsmønstre — systeminntrengning, sosial manipulering og webapplikasjonsangrep — sto for 96 prosent av alle innbrudd i små bedrifter. Rapporten fant også at 14 prosent av ansatte bruker generativ AI på bedriftens enheter, og at 72 prosent av disse gjør det via personlige e-postkontoer.

«Skygge-AI» er en tilknyttet utfordring. Software AG (2024) fant at 50 prosent av 6 000 kunnskapsarbeidere bruker uautoriserte AI-verktøy, og at 46 prosent ville fortsette selv om arbeidsgiveren forbød det. Bare 27 prosent kjører sikkerhetsskanning, og 29 prosent sjekker retningslinjer for datahåndtering. CybSafe og National Cybersecurity Alliance (2024) fant at 38 prosent deler sensitiv arbeidsinformasjon uten arbeidsgiverens kjennskap, og at 52 prosent mangler opplæring i sikker AI-bruk. JetBrains State of Developer Ecosystem 2024 bekrefter styringsvakuumet: nærmere 80 prosent av bedrifter tillater tredjepartens AI-verktøy eller mangler policy, mens bare 11 prosent forbyr bruk helt.

Li et al. (2024) fant i 26 intervjuer og 395 spørreundersøkelser at 71,6 prosent fryktet redusert kodekompetanse og 52,7 prosent fryktet tap av læringsevne. Erfarne utviklere var mer effektive i å formulere prompts, noe som skaper et kompetansegap som særlig rammer juniortunge småteam.

Småteam har likevel fordeler: GitHub Octoverse 2025 viser at nærmere 80 prosent av nye utviklere tar i bruk Copilot i løpet av første uke, og Cui et al. (2025) fant at mindre erfarne utviklere hadde høyere adopsjonsrate og større produktivitetsgevinst. Kort kommunikasjonsvei og rask beslutningstaking gjør det mulig å innføre nye verktøy og praksiser raskere enn i store organisasjoner.

2.11 Forskningsgap

Gjennomgangen av eksisterende litteratur avdekker et tydelig forskningsgap. Mens det finnes omfattende studier på AI-verktøyenes effekt på individuell utviklerproduktivitet i store organisasjoner som Microsoft, Google og Accenture (Cui et al., 2025; Paradis et al., 2024), er det vesentlig mindre forskning på hvordan disse verktøyene fungerer i konteksten av små team og oppstartsbedrifter.

Videre fokuserer mesteparten av den eksisterende litteraturen enten på produktivitet eller sikkerhet isolert sett. Det mangler studier som undersøker skjæringspunktet mellom

AI-akselerert utvikling og sikkerhetspraksis i ressursbegrensede team. Som Klemmer et al. (2024) påpeker, har utviklere ofte utilstrekkelige strategier for å verifisere sikkerheten i AI-generert kode, et problem som sannsynligvis forsterkes i småteam uten dedikerte sikkerhetsressurser. Apiiro (2025) formulerer dette som en bransjeobservasjon: «4x hastighet, 10x sårbarheter».

Det identifiserte forskningsgapet kan oppsummeres som: Lite forskning eksisterer på hvordan AI-verktøy påvirker produktivitet, kodekvalitet og sikkerhetspraksis i små utviklerteam med begrenset sikkerhetskompetanse. Denne oppgaven adresserer dette gapet ved å undersøke hvordan AI-akselerert utvikling kan integreres i smidige arbeidsflyter i små team, med fokus på å balansere produktivitet, sikkerhet og strukturell kvalitet.

3. Metode

Dette kapittelet beskriver hvordan studien er gjennomført. Det redegjøres for forskningsdesign og metodisk tilnærming, presentasjon av casen, datainnsamlingsmetoder, analyseprosess samt vurdering av validitet og begrensninger.

3.1 Forskningsdesign og tilnærming

Forskningsdesignet er forankret i Oates, Griffiths og McLean (2026), som beskriver hvordan ulike forskningsstrategier kan kombineres innenfor en samlet forskningsmetodologi. Studien har en hovedsakelig kvalitativ tilnærming med innslag av kvantitative data fra sikkerhetseksperimentet. Studien har et pragmatisk forskningsperspektiv, der valg av metoder er styrt av hva som best kan belyse problemstillingen. Samtidig har studien fortolkende trekk ved at forståelsen av AI-akselerert utvikling undersøkes gjennom erfaringer og praksis i en konkret organisatorisk kontekst. Formålet er ikke statistisk generalisering, men analytisk generalisering (Yin, 2018), hvor innsikten fra casen brukes til å utvikle begreper, forståelser og et praktisk rammeverk som kan ha overføringsverdi til lignende små utviklingsteam.

Oppgaven bruker to komplementære strategier:

Case Study er den overordnede strategien. Yin (2018) definerer en case-studie som en observasjonsbasert undersøkelse av et samtidsfenomen i sin virkelige kontekst, særlig når grensene mellom fenomen og kontekst er uklare. ToSee AS fungerer som en enkeltstående case der vi studerer AI-akselerert utvikling i en naturlig setting. Case-studien er primært deskriptiv

med forklarende elementer: vi beskriver hvordan AI-verktøy brukes i smidig utvikling, og forsøker å forklare hvilke faktorer som påvirker balansen mellom produktivitet og sikkerhet. I tråd med case-studie-metoden benytter vi flere datakilder: erfaringsdokumentasjon fra daglig bruk, et strukturert sikkerhetseksperiment, og utviklingsdokumentasjon som git-logger og sprint-notater.

Design and Creation er den andre strategien, brukt for å utvikle oppgavens praktiske bidrag: et rammeverk for AI-akselerert utvikling i småteam (presentert i Kapittel 5). Oates et al. (2026) beskriver Design and Creation som en strategi der forskeren utvikler en ny IT-artefakt, i dette tilfellet et rammeverk/modell som bidrag til kunnskap. Rammeverket er basert på erfaringsbaserte funn fra case-studien og følger den iterative prosessen bevissthet, forslag, utvikling, evaluering og konklusjon (Vaishnavi & Kuechler, 2015). Rammeverket tilhører kategorien «metode/modell» i March og Smiths (1995) typologi over IT-konstrukter.

Denne kombinasjonen er i tråd med Oates et al. (2026, kap. 8), som eksplisitt beskriver at Design and Creation kan brukes som én strategi innenfor en større forskningsmetodologi, for eksempel sammen med en case-studie. Case-studien gir det erfaringsbaserte grunnlaget, mens Design and Creation strukturerer det praktiske bidraget.

I tillegg inngår et kontrollert eksperiment (Oates et al., 2026, kap. 9) som en av datainnsamlingsmetodene innenfor case-studien. Sikkerhetseksperimentet tester AI-generert kode under kontrollerte betingelser og gir kvantitative data som supplerer de kvalitative erfaringene fra daglig bruk.

3.2 Casebeskrivelse: ToSee AS og prosjektkontekst

ToSee AS er et lite norsk AI-startup basert i Oslo som utvikler en plattform for text-to-speech (TTS) og syntetiske avatarvideoer. Selskapet har en liten kjerne med CTO Martin Bergo som ekstern veileder for denne oppgaven. Utviklingsteamet består av tre bachelorstudenter (forfatterne) som har deltatt som utviklere gjennom ett semester.

Prosjektet benytter en smidig arbeidsmetodikk uten et strengt formalisert rammeverk som Scrum eller Kanban, men med elementer fra begge, inkludert korte iterasjoner, hyppige leveranser og daglig kommunikasjon med CTO. Utviklingsarbeidet har vært organisert i ukentlige eller

to-ukentlige sprinter med definerte mål, og versjonskontroll har blitt håndtert gjennom Git med separate feature branches for hvert gruppemedlem, slik at arbeid kunne utføres parallelt uten å påvirke hovedkoden direkte. Når arbeid var ferdigstilt, ble kodeendringer integrert i utviklingsbranchen av teamlederen.

Teknologistakken består av React (frontend), Bun.js (backend) og SQLite (database). AI-verktøyene som har vært brukt i daglig utvikling inkluderer Claude Code (Opus 4.5/4.6), Opencode og Codex CLI (GPT-5.3-Codex). Disse verktøyene har vært benyttet til kodegenerering, feilretting, refaktorering og arkitektoniske beslutninger.

Valget av ToSee AS som case er basert på flere kriterier som Yin (2018) anbefaler for case-utvelgelse: tilgjengelighet (forfatterne hadde direkte tilgang som utviklere), representativitet (et typisk lite norsk tech-startup med begrensede ressurser), og unik mulighet (sjansen til å studere AI-akselerert utvikling fra innsiden i et reelt prosjekt).

3.3 Datainnsamlingsmetoder

I tråd med case-studie-metoden (Yin, 2018) benyttes flere datainnsamlingsmetoder for å oppnå triangulering og styrke funnenes troverdighet. Ved å kombinere kvalitative erfaringer fra daglig drift med kvantitative data fra et kontrollert eksperiment, kan vi fange opp både brukernes opplevelser og de tekniske realitetene ved AI-generert kode.

3.3.1 Erfaringsdokumentasjon fra daglig bruk

Gjennom hele utviklingsperioden har teamet ført en strukturert erfaringslogg der bruk av AI-verktøy er dokumentert. Loggen dekker hvilke verktøy som ble brukt, til hvilke oppgaver, opplevd effekt på produktivitet, observerte utfordringer, og eventuelle sikkerhets- eller kvalitetsproblemer i generert kode. Erfaringsdokumentasjonen fungerer som en forsker-generert datakilde (Oates et al., 2026, kap. 8) og gir kvalitativ innsikt i hvordan AI-verktøy påvirker daglig utviklingsarbeid i et lite team.

3.3.2 Sikkerhetseksperiment design og gjennomføring

For å undersøke sikkerhetskvaliteten i AI-generert kode under kontrollerte betingelser ble det gjennomført et strukturert eksperiment. Eksperimentet fungerer som et avgrenset kvantitativt

supplement til case-studien, der målet ikke er statistisk generalisering, men å undersøke konkrete forventninger basert på litteraturen om AI-generert kode og sikkerhetsrisiko. På bakgrunn av litteraturen formulert i kapittel 2 ble følgende hypoteser laget.

- **H1:** Sikkerhetsprompt vil gi høyere sikkerhetsscore enn nøytral prompt for AI-generert kode.
- **H2:** AI-generert kode fra nøytrale prompts vil oftere mangle grunnleggende sikkerhetstiltak, særlig knyttet til hemmelighetshåndtering, input-validering, passordlagring og sesjonshåndtering.
- **H3:** Statisk sikkerhetsanalyse med SAST vil ikke avdekke alle relevante sårbarheter i AI-generert kode, og må derfor suppleres med manuell gjennomgang.

Disse forventningene brukes ikke som grunnlag for statistisk hypotesetesting, ettersom utvalget er begrenset til 16 kodesampler. De fungerer i stedet som analytiske forventninger som strukturerer eksperimentdesignet og gjør det mulig å sammenligne resultatene med funnene fra litteraturstudien.

Eksperimentet følger et kombinert design med tre uavhengige variabler:

Modell: Fire AI-modeller ble testet: Claude Opus 4.6 (frontier-modell valgt fordi den representerer høyest tilgjengelig kvalitet og brukes aktivt i vår case gjennom Claude Code og er sammen med GPT-5.3-Codex, blant den mest brukte modellen), GPT-5.3-Codex (kodespesialist valgt fordi den er optimalisert for funksjonell koding og backend-logikk, og er sentral i utviklingsverktøy som Copilot/opencode), Gemini 3.1 Pro (bred tilgjengelighet valgt fordi den er mye brukt i praksis, særlig for frontend/UI-oppgaver, og representerer en modell mange utviklere har tilgang til), og Kimi K2.5 (kostnadseffektivt alternativ valgt for å inkludere en billigere modell og undersøke om lavere kost gir dårligere sikkerhetskvalitet). Utvalget representerer et spekter av tilgjengelige modeller med ulik posisjonering.

Oppgave: To typiske webutviklingsoppgaver ble brukt: (1) et innloggingssystem med brukerregistrering og autentisering i Flask/SQLite, og (2) et REST API for brukerprofiler med

JWT-autentisering. Begge oppgavene ble valgt fordi de inneholder flere kjente sikkerhetskritiske områder.

Prompttype: To prompttyper ble sammenlignet: (1) nøytral prompt med kun funksjonelle krav, og (2) sikkerhetsprompt som inkluderer eksplisitte sikkerhetskrav (beskyttelse mot SQL injection, passord-hashing, input-validering, hemmelighetsbehandling og rate limiting).

Dette gir et design på 4 modeller × 2 oppgaver × 2 prompttyper = 16 kodesampler totalt. Hver kodesample ble evaluert på seks sikkerhetskategorier med en skala fra 0 til 2 poeng per kategori, der 0 indikerer sårbar kode og 2 indikerer sikker implementasjon. Maksimal score er 12 poeng per sample. De seks kategoriene er:

Kategori	0 = Sårbar	2 = Sikker
SQL Injection	String concatenation	Parameteriserte queries
Passord	Plaintext/MD5	bcrypt/argon2
Input Validation	Ingen validering	Whitelist + type-sjekk
XSS	Ingen sanitering	Full escaping
Secrets	Hardkodet	Environment variables
Auth/Cookies	Usikker	httpOnly + Secure

Tabell 3.1: Evalueringskategorier og skala for sikkerhetseksperimentet.

3.3.3 Evalueringsverktøy (Semgrep og manuell gjennomgang)

Hver kodesample ble evaluert med to komplementære metoder:

Semgrep (SAST): Et open source-verktøy for statisk applikasjonssikkerhetstesting som automatisk skanner kildekode for kjente sårbarhetsmønstre. Semgrep ble kjørt på alle 16 samples med standard regelsett for Python/Flask.
Manuell gjennomgang: En sjekkliste-basert evaluering utført av forfatterne, der hver kodesample ble vurdert mot de seks sikkerhetskategoriene beskrevet i Tabell 3.1. Den manuelle gjennomgangen sikret at sårbarheter som SAST-verktøyet ikke fanger opp, som logiske feil, manglende funksjonalitet og subtile implementasjonsproblemer også ble identifisert. Kombinasjonen av automatisert og manuell evaluering er valgt for å undersøke hvor godt SAST alene fungerer som sikkerhetsverktøy for AI-generert kode, et spørsmål med direkte relevans for små team som mangler dedikerte sikkerhetsressurser.

3.4 Analyse og dokumentasjon av funn

Dataanalysen følger to spor som tilsvarer de to datatypene:

Kvantitativ analyse av sikkerhetseksperimentet: Resultatene aggregeres per modell, per prompttype og per sikkerhetskategori. Gjennomsnittsscore sammenlignes for å identifisere mønstre i hvordan ulike modeller og prompttyper påvirker sikkerhetskvaliteten i generert kode. Vanligste sårbarhetstyper rangeres etter frekvens.

Kvalitativ analyse av erfaringsdokumentasjonen: Erfaringsloggene analyseres tematisk for å identifisere gjennomgående mønstre, fordeler og utfordringer ved bruk av AI-verktøy i daglig utvikling. Denne tematiske analysen følger prinsippene beskrevet av Braun og Clarke (2006). Prosessen var iterativ, hvor teammedlemmene først kodet deler av erfaringsloggene individuelt. Deretter ble de innledende kodene sammenlignet og diskutert i fellesskap for å identifisere overordnede temaer og mønstre knyttet til produktivitet, sikkerhet, arkitektur og samarbeid. De endelige temaene ble definert og navngitt gjennom en konsensusbasert gjennomgang for å sikre at de var forankret i de erfaringsbaserte dataene. Funnene kategoriseres etter produktivitet, sikkerhet, arkitektur og samarbeid.

Resultatene fra begge analysesporene danner grunnlaget for rammeverket i Kapittel 5 og diskusjonen i Kapittel 6, der funnene knyttes tilbake til litteraturen og delspørsmålene.

3.5 Validitet, reliabilitet og begrensninger

Studiens interne validitet styrkes av at sikkerhetseksperimentet bruker et kontrollert design med definerte variabler, standardiserte prompts og en felles evalueringsskala på seks sikkerhetskategorier. Kombinasjonen av Semgrep og manuell gjennomgang gir metodetriangulering og reduserer risikoen for at sårbarheter overses. For erfaringsdokumentasjonen er forfatternes dobbeltrolle som både forskere og utviklere en mulig kilde til bias, men dette motvirkes ved at erfaringene ble dokumentert løpende og vurdert i fellesskap av teamet.

Den eksterne validiteten er begrenset fordi studien bygger på én case og ett utviklingsteam. Funnene er derfor ikke statistisk generaliserbare, men kan ha analytisk overføringsverdi til lignende små utviklingsteam som vurderer å ta i bruk AI-verktøy. Dette er i tråd med Walsham (1995), som vektlegger analytisk generalisering gjennom konsepter, implikasjoner og kontekstnær innsikt.

Reliabiliteten styrkes ved at eksperimentdesignet, evalueringsskjemaet og promptene er dokumentert. Samtidig er AI-modellers output ikke-deterministisk, noe som betyr at eksakte resultater ikke nødvendigvis kan replikeres. Studien bør derfor forstås som en kontekstbasert undersøkelse av mønstre, ikke som en statistisk generaliserbar måling.

4. Resultater

Dette kapittelet presenterer resultatene fra de to datasporene: sikkerhetseksperimentet (4.1) og erfaringene fra daglig bruk av AI-verktøy i ToSee-utviklingen (4.2). Sikkerhetseksperimentet gir kvantitative data om AI-generert kodes sikkerhetskvalitet, mens erfaringsdokumentasjonen gir kvalitativ innsikt i praktisk bruk over tid.

4.1 Sikkerhetseksperiment

Sikkerhetseksperimentet evaluerte 16 kodesampler generert av fire modeller på to oppgaver med to prompttyper. Hver sample ble scoret på seks sikkerhetskategorier (0 til 2 poeng per kategori, maks 12 poeng).

4.1.1 Resultater per modell

Tabell 4.1 viser totalscoren per modell fordelt på prompttype. Alle modeller viser en markant forbedring med sikkerhetsprompt.

Modell	Nøytral	Sikkerhet
Claude Opus 4.6	6.5 / 12	12 / 12
GPT-5.3-Codex	5 / 12	12 / 12
Gemini 3.1 Pro	5 / 12	11 / 12
Kimi K2.5	7 / 12	11.5 / 12
Gjennomsnitt	**5.875 / 12**	**11.625 / 12**

Tabell 4.1: Sikkerhetsscore per modell, fordelt på prompttype.

Gjennomsnittscoren øker fra 5.875 av 12 med nøytral prompt til 11.625 av 12 med sikkerhetsprompt. Den relative forbedringen er beregnet som $(11.625-5.875)/5.875 \approx 0.98$, altså omtrent 98 prosent relativ økning i sikkerhetsscore. Dette betyr at sikkerhetsprompten nesten dobler gjennomsnittlig score i dette eksperimentet. Claude Opus 4.6 og GPT-5.3-Codex oppnår maksimal score i evalueringsskjemaet (12/12) med sikkerhetsprompt. Selv de svakeste modellene på nøytral prompt, GPT-5.3-Codex og Gemini 3.1 Pro med 5/12, løftes til 12/12 og 11/12 med sikkerhetsprompt.

Et interessant funn er at Kimi K2.5 scorer høyest på nøytral prompt (7/12), noe som antyder at modellens standardinnstillinger inkluderer noe sikkerhetspraksis. Claude Opus 4.6 er likevel den mest konsistente modellen på tvers av begge prompttyper.

4.1.2 Effekt av sikkerhetsprompt

Tabell 4.2 viser resultater per sikkerhetskategori, fordelt på prompttype. Tallene angir hvor mange av 8 samples (4 modeller × 2 oppgaver) som oppfyller sikkerhetskravet.

Kategori	Nøytral (av 8)	Sikkerhet (av 8)
SQL Injection beskyttet	8 / 8	8 / 8
Passord korrekt hashet	3 / 8	8 / 8
Input validert	2 / 8	8 / 8
XSS beskyttet	1 / 8	8 / 8
Secrets i env vars	0 / 8	6 / 8
Auth/Cookies sikret	0 / 8	8 / 8

Tabell 4.2: Antall samples som oppfyller sikkerhetskrav per kategori og prompttype.

Resultatene avdekker et tydelig mønster. SQL Injection er det eneste området der alle modeller presterer godt også uten sikkerhetsprompt (8/8). Dette skyldes trolig at parameteriserte queries er sterkt representert i treningsdata og anses som standard praksis. I kontrast scorer secrets-håndtering og auth/cookies 0/8 på nøytral prompt, noe som betyr at ingen av modellene håndterer dette korrekt uten eksplisitte sikkerhetskrav. Sikkerhetsprompten løfter nesten alle kategorier til 8/8, med unntak av secrets som når 6/8. Det betyr at to modeller fortsetter å hardkode hemmeligheter selv med sikkerhetsprompt.

4.1.3 Vanligste sårbarheter i nøytrale prompts

Rangert etter frekvens er de vanligste sårbarhetene i AI-generert kode uten sikkerhetsprompt:

1. Hardkodede secrets (100 %): Alle 8 samples hardkoder API-nøkler, database-passord eller JWT-hemmeligheter direkte i kildekoden.

2. Usikker auth/cookies (100 %): Ingen modeller setter httpOnly- eller Secure-flagg på cookies, noe som gjør applikasjoner sårbare for session hijacking.

3. Manglende XSS-beskyttelse (87.5 %): I 7 av 8 samples saniteres ikke brukerinput før det inkluderes i HTML-output.

4. Manglende input-validering (75 %): I 6 av 8 samples aksepteres input fra brukere uten type-sjekk eller whitelist-validering.

5. Passord i plaintext (62.5 %): Over halvparten av samples lagrer passord uten hashing.

6. SQL Injection (0 %): Alle modeller bruker parameteriserte queries som standard.

Rangeringen viser at de mest grunnleggende sikkerhetspraksisene, som hemmelighetsbehandling og sikker sesjonshåndtering, er de som oftest mangler. Selv om det kan fremstå som et paradoks, viser resultatene at en kjent sårbarhetskategori som SQL injection håndteres bedre enn mer grunnleggende driftssikkerhetspraksis som hemmelighetshåndtering og sikre cookie-innstillinger. Dette antyder at modellenes treningsdata inneholder mye materiale om SQL injection-forebygging, mens driftssikkerhetspraksis som miljøvariabler og cookie-flagg er underrepresentert.

4.1.4 SAST vs. manuell gjennomgang

Tabell 4.3 viser en sammenlikning av sårbarheter oppdaget av Semgrep (SAST) og manuell gjennomgang.

Metode	Sårbarheter funnet
Semgrep (SAST)	2 / 16 samples
Manuell gjennomgang	12 / 16 samples

Tabell 4.3: Sårbarheter oppdaget per evalueringsmetode.

Forskjellen er påfallende. Semgrep fanget kun hardkodede secrets i 2 av 16 samples, men oppdaget ingen av tilfellene med passord lagret i plaintext, ingen manglende input-validering, ingen XSS-sårbarheter og ingen usikre cookie-konfigurasjoner. Manuell gjennomgang identifiserte sårbarheter i 12 av 16 samples, inkludert alle 8 samples generert med nøytral prompt og 2 samples generert med sikkerhetsprompt der hardkodede secrets fortsatt forekom.

Dette funnet har direkte implikasjoner for små team. SAST alene er utilstrekkelig som sikkerhetsverktøy for AI-generert kode. Automatisert skanning må suppleres med manuell gjennomgang, særlig for sårbarhetskategorier som logiske feil, manglende funksjonalitet og vanskelig sporbare konfigurasjonsfeil. For team uten dedikerte sikkerhetsressurser betyr dette at code review må inkludere en eksplisitt sikkerhetssjekkliste.

4.2 Erfaringer fra daglig bruk i ToSee-utviklingen

Denne seksjonen presenterer erfaringer fra daglig bruk av tre AI-verktøykonfigurasjoner i ToSee-utviklingen over perioden januar til april 2026. Erfaringene er basert på en strukturert erfaringslogg ført av teamet gjennom utviklingsperioden. Alle tre verktøyene er terminalbaserte og kjøres fra kommandolinje uten editor-integrasjon.

4.2.1 Opencode med GitHub Copilot-modeller

Opencode er et terminalbasert AI-kodingsverktøy med spesialiserte agenter per oppgavetype, konfigurert med modeller via GitHub Copilot-abonnement. Teamet benyttet et bredt spekter av modeller med definerte roller: Gemini 3.1 Pro for frontend og UI/UX, Claude-modeller (Opus 4.5/4.6 og Sonnet 4.5) for generelle oppgaver, GPT-5.3-Codex for analyse, og Gemini 3.1 Flash for dokumentasjon.

Opencode viste seg særlig sterk på design og UI/UX-oppgaver når riktig agent og modell ble valgt. Gemini 3.1 Pro genererte React-komponenter med god visuell struktur og god struktur i komponentene. Agentkonfigurasjonen, altså muligheten til å velge spesialiserte modeller per oppgavetype, ble opplevd som en klar styrke.

Svakhetene var mest tydelige i backend-logikk og funksjonalitet, der generert kode oftere måtte omskrives. Sikkerhetskvaliteten varierte mellom modellene. Kodekvaliteten ble vurdert som middels totalt sett, med middels behov for manuell justering. Verktøyet forsto ofte konteksten i den eksisterende kodebasen, men stabiliteten var noe varierende.

4.2.2 Claude Code med Opus 4.5/4.6

Claude Code er Anthropics CLI-verktøy for agentic koding, brukt med Claude Opus 4.5/4.6 gjennom hele utviklingsperioden. Verktøyet ble benyttet til React-komponenter, Node.js API-endepunkter, feilsøking og refaktorering.

Claude Code utmerket seg på flere områder. Feilsøking og forståelse av eksisterende kodebase ble vurdert som klart best av de tre verktøyene. Både frontend- og backend-kode holdt høy standard. Opus 4.5/4.6 viste seg særlig sterk på sikkerhet, med god håndtering av input-validering, autentisering og sensitiv data også uten sikkerhetsprompt. Kodekvaliteten ble vurdert som høy med lite behov for manuell justering.

Hovedbegrensningen var hastighet. Verktøyet var tregere på store oppgaver sammenlignet med Opencode, og kunne oppleves som overkill for enkle småoppgaver der raskere modeller ville holdt.

4.2.3 Codex CLI med GPT-5.3-codex

Codex CLI bruker OpenAIs GPT-5.3-codex modell og er spesialisert på funksjonell koding og logikk.

Verktøyet viste seg svært sterkt på funksjonalitet, backend-logikk og API-integrasjoner. Koden var gjennomgående ren, strukturert og velfungerende. Produktiviteten var høy på backend-oppgaver, og behovet for manuell justering var lite.

Svakheten lå i design og UI/UX, der frontendkode manglet ofte visuell kvalitet og måtte skrives om. Sikkerhetskvaliteten uten eksplisitte sikkerhetskrav var lav (5/12 i eksperimentet), med

hardkodede secrets, passord i plaintext og manglende input-validering. Med sikkerhetsprompt nådde verktøyet imidlertid 12/12.

4.2.4 Sammenligning av verktøy

Tabell 4.4 oppsummerer teamets vurdering av de tre verktøyene på tvers av ti kriterier.

Kriterium	Opencode	Claude Code	Codex CLI
Generell kodekvalitet	Middels	Høy	Høy
Design / UI-kvalitet	Høy	Høy	Middels
Funksjonalitet / logikk	Middels	Høy	Høy
Sikkerhet (u/prompt)	Middels	Høy	Lav
Sikkerhet (m/prompt)	Høy	Høy	Høy
Behov for gjennomgang	Middels	Lite	Lite
Workflow-integrasjon	Høy	Høy	Høy
Produktivitetseffekt	Høy	Høy	Høy
Kodebaseforståelse	Ofte	Ofte	Ofte
Pålitelighet / stabilitet	Middels	Høy	Høy

Tabell 4.4: Sammenlikning av AI-verktøy basert på teamets erfaringer.

Sammenlikningen viser at de ulike modell- og verktøykonfigurasjonene hadde komplementære styrker. I den praktiske ToSee-utviklingen ble både Claude Code, Codex CLI og Opencode prøvd ut, men Opencode ble etter hvert vurdert som det mest hensiktsmessige hovedverktøyet. Grunnen var at Opencode gjorde det mulig å arbeide terminalbasert med flere modeller i samme arbeidsflyt, blant annet Gemini 3.1 Pro, GPT/Codex- og Anthropic/Claude-modeller. Dette reduserte behovet for å bytte mellom flere separate CLI-verktøy, samtidig som teamet kunne velge modell etter oppgavetype.

Erfaringene viste at Gemini 3.1 Pro egnet seg best til design, frontend og UI/UX-relaterte oppgaver. Anthropic/Claude-modellene fungerte som de mest allsidige modellene og ga jevnt høy kodekvalitet, særlig ved feilsøking, refaktorering og arbeid med eksisterende kodebase. GPT/Codex-modellene var sterkest på funksjonalitet, backend-logikk og API-integrasjoner. Resultatene fra sikkerhetseksperimentet viste samtidig at alle modellene oppnådde høy sikkerhetsscore når sikkerhetsprompt ble brukt.

Et gjennomgående funn er derfor at bevisst valg av modell og prompt etter oppgavetype gir bedre resultater enn å bruke samme modelloppsett til alle typer arbeid. Teamet konkluderte med at en modellstrategi der sikkerhetskritiske oppgaver håndteres med Claude-modeller kombinert med eksplisitte sikkerhetsprompts og strukturerte code reviews, der design og frontendoppgaver støttes av Gemini 3.1 Pro, og funksjonelle backendoppgaver støttes av GPT/Codex-modeller, gir en god balanse mellom produktivitet, kodekvalitet og sikkerhet.

Et annet sentralt funn er at terminalbaserte AI-verktøy passer godt inn i teamets utviklingsflyt, men at Opencode ga den mest praktiske samlede arbeidsformen fordi flere modeller kunne brukes gjennom samme verktøy. AI-generert kode ble likevel ikke behandlet som ferdig kode, men ble systematisk gjennomgått som del av code review før merging. Denne praksisen, kombinert med bevisst bruk av sikkerhetsprompt for sensitiv kode, danner grunnlaget for anbefalingene i Kapittel 5.

5. Rammeverk for AI-akselerert utvikling i småteam

5.1 Grunnlag og formål

Dette kapittelet presenterer SAIF (Secure AI-Integrated Framework), et rammeverk for små utviklingsteam som ønsker å integrere AI-kodeverktøy i smidige arbeidsflyter uten å gå på akkord med sikkerhet, arkitektur eller pålitelighet. Rammeverket er utviklet som oppgavens primære artefakt i tråd med Design and Creation-strategien (Oates et al., 2026; March & Smith, 1995).

SAIF er bygget på tre grunnlag. Det første er erfaringsbaserte funn fra sikkerhetseksperimentet, som avdekket at sikkerhetsprompt alene løfter sikkerhetsscoren med 98 prosent, men at SAST kun fanger en brøkdel av sårbarhetene. Det andre er kvalitative erfaringer fra daglig bruk av AI-verktøy i ToSee-prosjektet, som viste at ulike modeller har komplementære styrker og at bevisst modellvalg gir bedre resultater. Det tredje er eksisterende litteratur om DevSecOps (IBM, u.å.; Anchore, 2025), AI-genererte sårbarheter (Pearce et al., 2022; Schreiber & Tippe, 2025) og produktivitetseffekter (Peng et al., 2023; Cui et al., 2025).

Rammeverket er designet for team på 2 til 8 personer som jobber smidig, bruker AI-kodeverktøy daglig, og ikke har dedikerte sikkerhetsroller. Det er modell- og verktøyagnostisk: selv om

eksemplene i dette kapittelet bruker verktøyene fra studien, er prinsippene anvendbare uavhengig av hvilke spesifikke AI-verktøy teamet bruker.

Utviklingen av SAIF tar utgangspunkt i fire hovedproblemer identifisert gjennom litteraturstudien, sikkerhetseksperimentet og erfaringsdokumentasjonen fra ToSee-prosjektet. For det første viser litteraturen at AI-kodeverktøy kan gi betydelige produktivitetsgevinster, men at effekten varierer med oppgavetype og kontekst (Peng et al., 2023; Cui et al., 2025; ThoughtWorks, 2025). Dette adresseres gjennom beslutningsmodellen for modellvalg. For det andre viste sikkerhetseksperimentet at AI-generert kode uten eksplisitte sikkerhetskrav ofte mangler grunnleggende sikkerhetstiltak, særlig knyttet til hemmelighetshåndtering, passordlagring, input-validering og sesjonshåndtering. Dette adresseres gjennom sikkerhetsnivåsystemet og obligatorisk sikkerhetsprompt for sikkerhetskritisk kode. For det tredje viste eksperimentet at SAST alene ikke avdekket alle relevante sårbarheter, noe som begrunner behovet for en strukturert sikkerhetssjekkliste i code review. For det fjerde viste erfaringene fra ToSee at AI-verktøy må integreres i eksisterende smidige arbeidsflyter for å gi verdi uten å skape mer koordinasjonsarbeid eller teknisk gjeld. Dette adresseres gjennom arbeidsflytmodellen og adopsjonstrappen.

SAIF består av fem komponenter som til sammen dekker hele utviklingssyklusen: (1) en beslutningsmodell for modellvalg, (2) et sikkerhetsnivåsystem med tre nivåer, (3) en sikkerhetssjekkliste for code review, (4) en arbeidsflytmodell for sprint-syklusen, og (5) et adopsjonstrappesystem for gradvis innføring.

5.2 Komponent 1: Beslutningsmodell for modellvalg

Et gjennomgående funn i studien er at ulike AI-modeller har komplementære styrker, og at bevisst valg av modell per oppgavetype gir bedre resultater enn å bruke én modell til alt. Dette fremgår særlig av sammenligningen i Tabell 4.4 og erfaringene fra ToSee-utviklingen, der Gemini 3.1 Pro, brukt gjennom Opencode, viste seg særlig sterk på design og UI-relaterte oppgaver, Claude Opus 4.5/4.6, brukt gjennom Claude Code, fremsto som den mest allsidige og sikkerhetssterke modellen, og GPT-5.3-codex, brukt gjennom Codex CLI, var sterkest på backend-logikk, funksjonalitet og API-integrasjoner.

Beslutningsmodellen er derfor utviklet som et svar på et erfaringsbasert funn fra ToSee-casen: produktivitetsgevinsten fra AI-verktøy øker når modellvalg tilpasses oppgavetype og risiko. Verktøyene fungerer i denne sammenhengen som arbeidsflater for modellene, mens selve beslutningen handler om hvilken modellprofil som passer best til oppgaven. Dette er også i tråd med litteraturen, som viser at effekten av AI-verktøy varierer med oppgavetype, utviklerkontekst og kompleksitet (Peng et al., 2023; Cui et al., 2025; ThoughtWorks, 2025). Beslutningsmodellen formaliserer dette funnet til en prosess som andre team kan følge.

5.2.1 Prinsipp

Beslutningsmodellen er basert på at hver utviklingsoppgave kan klassifiseres langs to akser: oppgavetype (hva som skal bygges) og sikkerhetssensitivitet (hvor stor risiko feil kode utgjør). Kombinasjonen av disse to aksene bestemmer hvilken type modell som bør velges og hvilke sikkerhetstiltak som kreves.

5.2.2 Beslutningsflyt

Når en utvikler starter en oppgave, følges denne beslutningsflyten:

Steg 1: Klassifiser oppgavetypen. Er dette primært en frontend/design-oppgave, en backend/logikk-oppgave, eller en sikkerhetskritisk oppgave (autentisering, brukerdata, ekstern input, hemmeligheter, API-integrasjon med sensitiv data)?

Steg 2: Bestem sikkerhetsnivå. Basert på klassifiseringen tildeles oppgaven et av tre sikkerhetsnivåer (beskrevet i detalj i komponent 2): Nivå 1 (sikkerhetskritisk), Nivå 2 (forretningslogikk) eller Nivå 3 (presentasjon).

Steg 3: Velg modellprofil. Basert på oppgavetype og sikkerhetsnivå velges en modell med dokumenterte styrker for denne kombinasjonen. Teamet bør tidlig i prosjektet kartlegge hvilke modeller de har tilgang til og teste dem på representative oppgaver.

Steg 4: Aktiver passende sikkerhetstiltak. Sikkerhetsnivået bestemmer hvilke tiltak som kreves (sikkerhetsprompt, code review, SAST). Se komponent 2.

5.2.3 Modellprofilkatalog

For at beslutningsmodellen skal være praktisk, bør teamet bygge en modellprofilkatalog: et enkelt dokument som beskriver hvilke modeller teamet har tilgang til og hva de er best egnet til. Katalogen bør oppdateres etter hvert som teamet samler erfaringer. Tabell 5.1 viser et eksempel basert på denne studien.

Profil	Egnet til	Sikkerhet u/prompt	Sikkerhet m/prompt	Anbefalt for
Claude Code (Opus 4.6)	Allsidig, feilsøking, refaktorering	Høy (6.5/12)	Perfekt (12/12)	Nivå 1 og Nivå 2
Gemini 3.1 Pro	Frontend, UI/UX, komponentdesign	Middels (5/12)	Høy (11/12)	Nivå 3
Codex (GPT-5.3-Codex)	Backend-logikk, API-er, funksjonalitet	Lav (5/12)	Perfekt (12/12)	Nivå 2 med prompt

Tabell 5.1: Eksempel på modellprofilkatalog fra ToSee-prosjektet.

Katalogen er ment som et levende dokument. Når nye modeller blir tilgjengelige eller teamet oppdager nye styrker og svakheter, bør katalogen oppdateres. Team som starter uten egne testdata kan bruke denne studiens resultater som utgangspunkt og tilpasse basert på egne erfaringer.

5.3 Komponent 2: Sikkerhetsnivåsystem

Sikkerhetsnivåsystemet er utviklet som svar på funnene fra sikkerhetseksperimentet og litteraturen om AI-genererte sårbarheter. Eksperimentet viste at nøytrale prompts ga systematiske sikkerhetssvakheter, blant annet hardkodede hemmeligheter, usikker sesjonshåndtering, manglende input-validering og svak passordhåndtering. Samtidig viste resultatene at sikkerhetsprompt økte gjennomsnittlig sikkerhetsscore fra 5.875 til 11.625 av 12. Dette støtter litteraturen som peker på at AI-generert kode ofte ser funksjonelt korrekt ut, men kan inneholde skjulte sikkerhetsfeil (Pearce et al., 2022; Klemmer et al., 2024; Schreiber & Tippe, 2025).

Ikke all kode krever samme grad av sikkerhetstiltak. Et sentralt problem i eksisterende litteratur om DevSecOps er at anbefalingene ofte presenteres som alt eller ingenting. For små team med begrensede ressurser fører dette i praksis til at sikkerhet nedprioriteres fordi den opplevde

kostnaden er for høy. Sikkerhetsnivåsystemet løser dette ved å differensiere. Tre nivåer gjør det mulig å konsentrere ressursene der de gir mest effekt.

5.3.1 Nivå 1: Sikkerhetskritisk kode

Nivå 1 gjelder kode som håndterer autentisering og autorisasjon, passord og hemmeligheter, brukerdata og personopplysninger, ekstern input (API-mottak, skjemaer, URL-parametere), sesjonshandtering og cookies, samt database-spørringer med brukerdata.

Krav: Sikkerhetsprompt er obligatorisk. Modell med høy baseline-sikkerhet bør velges. Code review med sikkerhetssjekkliste (komponent 3) er obligatorisk. SAST-skanning kjøres automatisk. Manuell gjennomgang fokuserer spesielt på de seks kategoriene fra eksperimentet.

Begrunnelse: Sikkerhetseksperimentet viste at 100 prosent av samples uten sikkerhetsprompt hadde hardkodede secrets og usikre cookies. Disse sårbarhetene oppstår konsekvent i nettopp denne typen kode.

5.3.2 Nivå 2: Forretningslogikk

Nivå 2 gjelder kode som implementerer forretningsregler, databehandling og transformasjoner, API-integrasjoner uten sensitiv data, samt intern applikasjonslogikk.

Krav: Sikkerhetsprompt er valgfri, men anbefalt ved tvil. Code review gjennomføres med fokus på funksjonell korrekthet og kodestruktur. SAST-skanning kjøres automatisk. Manuell sikkerhetsgjennomgang er ikke obligatorisk, men reviewer bør være oppmerksom på uventet håndtering av data.

5.3.3 Nivå 3: Presentasjon og UI

Nivå 3 gjelder ren frontend-kode som visuell layout, styling og komponentstruktur, statisk innhold og presentasjon, samt UI-interaksjoner uten brukerdata.

Krav: Sikkerhetsprompt er ikke nødvendig. Code review fokuserer på visuell kvalitet, komponentstruktur og gjenbrukbarhet. SAST kjøres automatisk som del av pipeline. Manuell sikkerhetsgjennomgang er ikke nødvendig.

5.3.4 Nivåmatrise

Tiltak	Nivå 1 (Kritisk)	Nivå 2 (Logikk)	Nivå 3 (UI)

Sikkerhetsprompt	Obligatorisk	Anbefalt	Ikke nødvendig
Modell med høy baseline	Obligatorisk	Valgfri	Valgfri
Code review	Oblig. m/sjekkliste	Obligatorisk	Anbefalt
Manuell sikkerhetsgjennomgang	Obligatorisk	Ved behov	Ikke nødvendig
SAST i CI/CD	Obligatorisk	Obligatorisk	Obligatorisk
Secret scanning	Obligatorisk	Obligatorisk	Obligatorisk

Tabell 5.2: Sikkerhetsnivåmatrise.

5.4 Komponent 3: Sikkerhetssjekkliste for code review

Sikkerhetssjekklisten er utviklet som svar på et tydelig funn i sikkerhetseksperimentet: Semgrep (SAST) oppdaget kun sårbarheter i 2 av 16 samples, mens manuell gjennomgang identifiserte problemer i alle 16. Dette viser at automatisert sikkerhetsanalyse alene ikke er tilstrekkelig for å evaluere AI-generert kode, særlig når svakhetene fører til kode som ikke er sikker, for eksempel knyttet til logikkfeil, feilkonfigurasjon, manglende validering eller utrygg håndtering av autentisering. Funnet støtter litteraturen som anbefaler at AI-generert kode behandles som kode som krever eksplisitt review og verifikasjon før integrering (Tambon et al., 2025; Kashif et al., 2025).

Sjekklisten bygger direkte på de seks sikkerhetskategoriene som ble brukt i eksperimentdesignet i kapittel 3.3.2: SQL injection, passord, input-validering, XSS, secrets og auth/cookies. Den fungerer dermed som en praktisk oversettelse av evalueringskriteriene fra metodekapittelet til en konkret review-praksis for småteam. Manuell review er derfor uunnværlig, men den må være strukturert for å være effektiv. Denne sjekklisten er designet for å brukes ved code review av AI-generert kode på Nivå 1 og Nivå 2.

5.4.1 Sjekklisten

Sjekklisten er organisert etter de seks sikkerhetskategoriene fra eksperimentet, utvidet med konkrete kontrollspørsmål som revieweren skal besvare.

Kategori	Kontrollspørsmål	OK?

SQL Injection	Bruker alle database-spørringer parameteriserte queries? Settes brukerinput aldri direkte inn i SQL-strenger?	□ Ja □ Nei
Passord	Hashes passord med bcrypt, argon2 eller tilsvarende? Lagres passord aldri i plaintext eller med svak hashing (MD5, SHA1)? Finnes minimum passordlengde?	□ Ja □ Nei
Input-validering	Valideres all ekstern input med type-sjekk og lengdebegrensning? Brukes whitelist-tilnærming der det er mulig? Avvises ugyldig input med tydelig feilmelding?	□ Ja □ Nei
XSS	Saniteres all brukerinput før inkludering i HTML-output? Brukes template-rammeverk med auto-escaping? Settes Content-Security-Policy-header?	□ Ja □ Nei
Secrets	Lagres alle hemmeligheter i miljøvariabler eller secret manager? Finnes det ingen hardkodede hemmeligheter i kildekoden? Er .env-filer lagt til i .gitignore?	□ Ja □ Nei
Auth/Cookies	Settes httpOnly- og Secure-flagg på cookies? Brukes SameSite-attributt? Har sesjoner fornuftig utløpstid? Er autorisasjonssjekker implementert på alle beskyttede endepunkter?	□ Ja □ Nei

Tabell 5.3: Sikkerhetssjekkliste for code review av AI-generert kode.

5.4.2 Bruk av sjekklisten

Sjekklisten er designet for å kunne gjennomføres på under fem minutter per pull request. Revieweren går gjennom hver kategori og besvarer kontrollspørsmålene. Dersom svaret er «Nei» på noen av spørsmålene, må koden rettes før den merges.

For Nivå 1-kode bør alle seks kategorier gjennomgås. For Nivå 2-kode er det tilstrekkelig å fokusere på de kategoriene som er relevante for den spesifikke oppgaven. Sjekklisten bør være tilgjengelig som et fast element i teamets pull request-mal, slik at den ikke glemmes.

Et sentralt poeng er at sjekklisten ikke erstatter generell code review, men supplerer den med et sikkerhetsrettet perspektiv. Revieweren bør fortsatt vurdere kodekvalitet, lesbarhet, arkitektonisk konsistens og funksjonell korrekthet i tillegg til sikkerhetskategoriene.

5.5 Komponent 4: Arbeidsflytmodell for sprint-syklusen

Arbeidsflytmodellen er utviklet for å koble SAIF-komponentene til den smidige arbeidsformen som ble observert i ToSee-prosjektet. Erfaringene fra case-studien viste at AI-verktøy ga størst verdi når de ble integrert i eksisterende utviklingspraksiser som feature-brancher, pull requests, code review og korte iterasjoner. Samtidig viste litteraturen at AI-akselerert utvikling kan øke både utviklingstempo og risiko for teknisk gjeld dersom generert kode aksepteres ukritisk (Moreschini et al., 2026; GitClear, 2025; OX Security, 2025).

For at beslutningsmodellen, sikkerhetsnivåsystemet og sjekklisten skal fungere i praksis, må de derfor integreres i teamets eksisterende sprint-syklus. Arbeidsflytmodellen beskriver hvordan AI-verktøy og sikkerhetstiltak passer inn i en typisk smidig arbeidsflyt for småteam.

5.5.1 Sprint-syklusen med AI

Modellen er basert på en forenklet sprint-syklus med fem faser. For hver fase beskrives hvilke AI-relaterte aktiviteter og sikkerhetstiltak som bør gjennomføres.

Fase	AI-aktiviteter	Sikkerhetstiltak
1. Planlegging	Klassifiser oppgaver etter type og sikkerhetsnivå. Velg modellprofil per oppgave via beslutningsmodellen.	Tildel sikkerhetsnivå (1, 2 eller 3) til hver oppgave. Merk oppgaver som krever sikkerhetsprompt.
2. Utvikling	Generer kode med valgt modell. Bruk sikkerhetsprompt for Nivå 1-oppgaver. Bytt modell ved behov.	Følg modellprofilkatalogen. Dokumenter hvilken modell og prompttype som ble brukt per oppgave.
3. Code review	Reviewer vurderer AI-generert kode på lik linje med manuelt skrevet kode.	Bruk sikkerhetssjekklisten (Tabell 5.3) for Nivå 1- og Nivå 2-kode. Blokker merging ved ubesvarte kontrollspørsmål.
4. Integrasjon	Merge godkjent kode til hovedgren via pull request.	SAST og secret scanning kjøres automatisk i CI/CD. Blokker merge ved kritiske funn.
5. Retrospektiv	Evaluer modellvalg og AI-verktøybruk. Oppdater modellprofilkatalogen. Del erfaringer i teamet.	Gjennomgå sikkerhetsfunn fra sprinten. Identifiser gjentakende mønstre. Juster sikkerhetsnivåer ved behov.

Tabell 5.4: Sprint-syklusen med AI-aktiviteter og sikkerhetstiltak per fase.

5.5.2 Versjonskontroll og parallelt arbeid

Små team som bruker AI-verktøy produserer potensielt mer kode raskere, noe som øker behovet for god versjonskontrollpraksis. En effektiv tilnærming er at hver utvikler arbeider i sin egen branch, og at teamleder merger godkjente endringer til hovedgrenen via utviklere sine pull requests. Dette isolerer endringer og reduserer konflikter når flere bruker AI-verktøy samtidig. Atlassian (u.å.) poengterer at riktig bruk av versjonskontroll isolerer endringer, muliggjør parallelt arbeid og reduserer risiko knyttet til sammenslåing.

5.5.3 Håndtering av teknisk gjeld

AI-akselerert utvikling kan potensielt øke teknisk gjeld dersom generert kode aksepteres ukritisk (Moreschini et al., 2026). Rammeverket anbefaler jevnlige refaktoreringssprinter. Farry (2026) foreslår en tilnærming der mennesker definerer de stabile rammeverkene og lar AI fylle inn forretningslogikk, slik at tversgående krav som logging, autentisering og feilhåndtering beholdes under menneskelig kontroll.

5.6 Komponent 5: Adopsjonstrappesystem

Rammeverkets adopsjonstrapper er utviklet som svar på utfordringene småteam har med begrensede ressurser, manglende dedikerte sikkerhetsroller og behov for rask iterasjon. Litteraturgjennomgangen viste at små virksomheter ofte har større relativ risiko og færre sikkerhetsressurser enn større organisasjoner (Ji et al., 2024; Verizon Business, 2025), samtidig som de har mye å vinne på AI-akselerert utvikling. En full DevSecOps-prosess kan derfor være urealistisk å innføre på én gang.

Ikke alle team er klare for å innføre hele rammeverket samtidig. Adopsjonstrappesystemet gir derfor en gradvis innføringsplan i fire trinn, der hvert trinn bygger på det forrige. Målet er at team kan starte med de mest effektive lavterskeltiltakene, særlig sikkerhetsprompt og code review, før de gradvis innfører mer struktur og automatisering.

5.6.1 Trinn 1: Grunnmur (uke 1 til 2)

Mål: Etablere de mest effektive enkelttiltakene med minimal investering.

Tiltak: Definer en standard sikkerhetsprompt som teamet bruker for all sikkerhetskritisk kode. Aktiver secret scanning på repository-nivå (gratis i GitHub og GitLab). Innfør regelen om at all AI-generert kode skal gå gjennom code review før merging.

5.6.2 Trinn 2: Strukturering (uke 3 til 4)

Mål: Innføre systematikk i modellvalg og sikkerhetstiltak.

Tiltak: Klassifiser eksisterende og nye oppgaver etter sikkerhetsnivå (Nivå 1, 2 eller 3). Bygg en modellprofilkatalog basert på teamets erfaringer så langt. Ta i bruk sikkerhetssjekklisten som fast element i pull request-malen.

5.6.3 Trinn 3: Automatisering (uke 5 til 8)

Mål: Automatisere det som kan automatiseres, slik at manuell innsats fokuseres der den gir mest verdi.

Tiltak: Integrer SAST (f.eks. Semgrep) i CI/CD-pipelinen slik at det kjøres automatisk ved hver commit. Legg til SCA (Software Composition Analysis) for å skanne avhengigheter. Konfigurer automatisk blokkering av merge ved kritiske SAST-funn.

5.6.4 Trinn 4: Kontinuerlig forbedring (løpende)

Mål: Etablere en kultur for læring og forbedring av AI-verktøybruk.

Tiltak: Gjennomfør retrospektiv med AI-fokus ved slutten av hver sprint. Oppdater modellprofilkatalogen basert på nye erfaringer. Planlegg jevnlige refaktoreringssprinter for å håndtere teknisk gjeld. Del erfaringer og mønstre på tvers av teamet. Vurder nye modeller og verktøy når de blir tilgjengelige.

5.6.5 Adopsjonsoversikt

Trinn	Tidsramme	Hovedtiltak	SAIF-komponenter
1: Grunnmur	Uke 1 til 2	Sikkerhetsprompt, secret scanning, code review	Deler av komponent 2 og 3
2: Strukturering	Uke 3 til 4	Sikkerhetsnivåer, modellkatalog, sjekkliste i PR	Komponent 1, 2 og 3 komplett

3: Automatisering	Uke 5 til 8	SAST i CI/CD, SCA, automatisk blokkering	Komponent 4 (integrasjon)
4: Forbedring	Løpende	Retrospektiv, katalogoppdatering, refaktorering	Komponent 4 og 5 komplett

Tabell 5.5: Adopsjonstrappesystem for gradvis innføring av SAIF.

5.7 Oppsummering av SAIF

SAIF tilbyr små team et strukturert rammeverk for sikker og produktiv AI-integrering i smidige arbeidsflyter. De fem komponentene kan brukes separat eller samlet; et team kan for eksempel starte med sikkerhetstiltak og gradvis utvide via adopsjonstrappen.

Rammeverket er basert på studiens observasjoner, men designet for generalisering. Det inkluderer oversikt over sikkerhetsnivåer, sjekklister og en modellprofilkatalog. Komponentene er verktøy-uavhengige og tilpasses ulike teknologistakker og team-modenhet.

En hovedstyrke er at SAIF ikke krever egne sikkerhetsroller; tiltakene utføres av utviklere som del av daglig drift. Sjekklisten tar under fem minutter per review, og sikkerhetsprompt krever minimal innsats, noe som gjør rammeverket realistisk selv for team med svært begrensede ressurser.

6. Diskusjon

Dette kapittelet drøfter funnene fra studien i lys av eksisterende litteratur, besvarer delspørsmålene, vurderer studienes begrensninger, og diskuterer praktiske implikasjoner for små utviklingsteam.

6.1 Funn opp mot litteraturen

6.1.1 Produktivitet og AI-verktøy

Erfaringene fra ToSee-prosjektet peker i samme retning som litteraturen om at AI-verktøy kan øke utviklerproduktiviteten. Alle tre verktøykonfigurasjonene ble vurdert til å gi høy produktivitetseffekt i den daglige arbeidsflyten. Dette samsvarer med Peng et al. (2023), som fant at utviklere med GitHub Copilot fullførte en oppgave 55,8 prosent raskere enn

kontrollgruppen, og Cui et al. (2025), som rapporterte en 26,08 prosent økning i fullførte oppgaver på tvers av tre feltstudier.

Et viktig supplement til denne litteraturen er at våre erfaringer nyanserer produktivitetsbegrepet. Den observerte effektivitetsøkningen var ikke jevnt fordelt på tvers av oppgavetyper. I tråd med ThoughtWorks (2025), som påpeker at gevinster er særlig tydelige i repetitive oppgaver og dokumentasjon mens mer komplekse flerfil-kontekster gir svakere effekter, fant vi også at AI-verktøyene var sterkest på avgrenset kodegenerering og svakere på oppgaver som krevde dypere forståelse av hele systemarkitekturen.

Et funn som i liten grad er dekket i den eksisterende litteraturen er verdien av bevisst modellvalg. De fleste studier tester én modell isolert (typisk GitHub Copilot). Våre erfaringer viser at små team kan oppnå bedre resultater ved å kombinere flere modeller med komplementære styrker, noe som underbygger behovet for oversikt over modellprofiler i SAIF-rammeverket.

6.1.2 Sikkerhetssårbarheter i AI-generert kode

Sikkerhetseksperimentet bekrefter og utdyper funnene fra Pearce et al. (2022), som rapporterte at 40 prosent av Copilot-generert kode inneholdt sårbarheter i sikkerhetssensitive scenarioer. Våre resultater viser en tilsvarende trend: uten sikkerhetsprompt scorer AI-generert kode i gjennomsnitt 5.875 av 12 på sikkerhetsevaluering, med 100 prosent forekomst av hardkodede secrets og usikre cookies.

Funnene våre støtter også Klemmer et al. (2024), som beskriver en «illusjon av korrekthet» der utviklere stoler på AI-generert kode uten tilstrekkelig verifikasjon. I vårt eksperiment produserte alle modeller kode som ser funksjonelt korrekt ut, men som inneholder systematiske sikkerhetssvakheter som ikke er synlige uten målrettet gjennomgang.

Et nytt bidrag fra vår studie er den detaljerte rangeringen av sårbarhetstyper. Mens Schreiber og Tippe (2025) og Cloud Security Alliance (2025) gir aggregerte statistikker om andelen usikker AI-generert kode, gir våre resultater en mer finkornet analyse av hvilke sikkerhetskategorier som er mest utsatt. Funnet om at SQL injection håndteres godt (0 prosent sårbar) mens secret-håndtering svikter konsekvent (100 prosent sårbar) gir praktisk veiledning for hvor små team bør fokusere sin manuelle gjennomgang.

6.1.3 Effekten av sikkerhetsprompt

Det kanskje mest påfallende funnet i studien er den dramatiske effekten av sikkerhetsprompt: Den relative forbedringen er beregnet som $(11.625-5.875)/5.875 \approx 0.98$, altså omtrent 98 prosent relativ økning i sikkerhetsscore. Dette er et deskriptivt mål på forskjellen mellom de to betingelsene i dette eksperimentet, og må ikke tolkes som en statistisk generaliserbar effektstørrelse. Denne effektstørrelsen overskrider det meste av det som er rapportert i eksisterende litteratur. Ji et al. (2024) identifiserer promptkvalitet som en faktor som påvirker sikkerhet i AI-generert kode, men kvantifiserer ikke effekten på samme måte som vårt eksperiment.

Funnet har viktige implikasjoner for debatten om AI-generert kodes sikkerhet. Mye av litteraturen presenterer et bilde der AI-generert kode er iboende usikker (Veracode, 2025; Apiiro, 2025). Våre resultater nyanserer dette: problemet er ikke at AI-modeller ikke kan generere sikker kode, men at de ikke gjør det med mindre de blir bedt om det. Sikkerhetskvaliteten er i stor grad en funksjon av promptkvaliteten.

6.1.4 SAST som sikkerhetsverktøy for AI-generert kode

Funnet om at Semgrep kun oppdaget sårbarheter i 2 av 16 samples er bekymringsverdig i lys av anbefalinger i litteraturen om å integrere SAST i CI/CD-pipelinen (Kalva, 2024; Anchore, 2025). Schreiber og Tippe (2025) brukte CodeQL i sin studie og identifiserte 4 241 CWE-instanser i AI-generert kode, noe som antyder at valg av SAST-verktøy og regelsett har stor betydning for deteksjonsraten.

Vårt funn betyr ikke at SAST er verdiløst, men at team må være bevisste på verktøyets begrensninger. For små team som mangler dedikerte sikkerhetsressurser er dette særlig relevant: dersom teamet stoler på SAST som primært sikkerhetstiltak, vil de fleste sårbarhetene i AI-generert kode passere uoppdaget. SAIF-rammeverkets kombinasjon av sikkerhetsprompt, manuell review med sjekkliste og SAST adresserer dette ved å skape flere lag med forsvar.

6.1.5 DevSecOps i småteam

Litteraturen om DevSecOps (IBM, u.å.; Wiz, 2026; Bright Security, 2025) anbefaler shift-left-prinsipper og integrert sikkerhet i livssyklusen for utvikling. Våre erfaringer fra ToSee bekrefter at disse prinsippene er gjennomførbare også for små team, men at de må tilpasses. Det

tradisjonelle DevSecOps-rammeverket forutsetter ofte dedikerte sikkerhetsroller, omfattende verktøykjeder og formaliserte prosesser som ikke er realistiske for et team på tre personer.

SAIF-rammeverkets sikkerhetsnivåsystem er et forsøk på å operasjonalisere DevSecOps-prinsipper for småteam ved å differensiere innsatsen basert på risiko. I stedet for å kreve samme sikkerhetstiltak for all kode, konsentrerer nivåsystemet ressursene der risikoen er høyest. Denne tilnærmingen er konsistent med Anchores (2025) anbefaling om risikobasert prioritering.

6.2 Delspørsmål besvart

6.2.1 DS1: Systematisk implementering av AI-verktøy for sikkerhet, arkitektur og pålitelighet

Delspørsmål 1 spør hvordan AI-verktøy kan implementeres systematisk for å ivareta cybersikkerhet, robust arkitektur og systempålitelighet i små team.

Studien viser at systematisk implementering krever tre elementer: bevisst modellstrategi basert på dokumenterte styrker per oppgavetype, sikkerhetsprompt som standardpraksis for sensitiv kode (som alene løfter sikkerhetskvaliteten med 98 prosent), og strukturerte valideringsmekanismer i form av sikkerhetssjekkliste og automatisert SAST. Ingen av mekanismene er tilstrekkelige alene, men de gir tilsammen et robust forsvar. SAIF-rammeverkets tre første komponenter formaliserer dette.

6.2.2 DS2: Avveininger mellom sikkerhet og smidighet, og støtte for parallelt arbeid

Delspørsmål 2 spør hvordan man veier avveininger mellom sikkerhet og smidighet, og hvilke metoder som støtter parallelt arbeid.

Et sentralt funn er at sikkerhet og smidighet ikke nødvendigvis står i motsetning når tiltakene er riktig tilpasset. Sikkerhetsprompten krever ingen ekstra tid, og sikkerhetssjekklisten i code review legger til noen minutter per merge. SAIF-rammeverkets nivåsystem sørger for at høy sikkerhetsinnsats kun kreves for Nivå 1-kode, slik at sikkerhet ikke blir en flaskehals. For parallelt arbeid viste erfaringene fra ToSee at opplegget med individuelle brancher mot en felles

integrasjonsmiljø, med teamleder som merger til hovedgrenen, fungerer godt også med aktiv AI-verktøybruk

6.2.3 DS3: AI-teknologier og arbeidsmetoder for innovasjon i en cybersikkerhetskontekst

Delspørsmål 3 spør hvordan moderne AI-teknologier og arbeidsmetoder kan øke innovasjon og produktutvikling i en cybersikkerhetskontekst.

Erfaringene fra ToSee viser at AI-verktøy kan fungere som en muliggjører for innovasjon: ved å automatisere repetitive oppgaver frigjøres kapasitet til mer kreativt arbeid, og kostnaden ved å prototype en idé blir vesentlig lavere (ThoughtWorks, 2025). Samtidig krever innovasjon i en cybersikkerhetskontekst disiplin. AI-verktøy kan bidra til at svakheter i utviklingspraksis forsterkes, og uten strukturerte sikkerhetstiltak kan økt utviklingstempo føre til at sårbarheter akkumuleres raskere enn de blir identifisert (Apiiro, 2025). SAIF-rammeverkets adopsjonstrapp løser dette ved at teamet kan begynne å innovere med AI-verktøy umiddelbart, fra trinn 1, uten å vente på at hele sikkerhetsprosessen er på plass.

6.3 Begrensninger ved studien

Studien har flere begrensninger som må tas i betraktning ved tolkning av resultatene. For det første er sikkerhetseksperimentet basert på 16 kodesampler og én kjøring per kombinasjon. Dette gir et begrenset grunnlag for statistiske konklusjoner, og resultatene bør derfor tolkes som indikasjoner på mønstre, ikke som endelige bevis for forskjeller mellom modeller eller prompttyper. Siden AI-modellers output er ikke-deterministisk, kan resultatene også variere ved replikering.

For det andre bygger erfaringsdokumentasjonen på ett team i én bedrift. Funnene reflekterer dermed en spesifikk organisatorisk og teknologisk kontekst, med ToSee AS, React/Bun.js i prosjektarbeidet og Python/Flask i sikkerhetseksperimentet. Overføring til andre team, teknologistakker eller organisasjoner krever derfor tilpasning.

For det tredje har forfatterne hatt en dobbeltrolle som både forskere og utviklere. Dette ga nærhet til praksisfeltet og detaljert innsikt i arbeidsflyten, men kan også ha påvirket tolkningen av erfaringene. Løpende erfaringslogg og et kontrollert sikkerhetseksperiment ble brukt for å redusere denne risikoen.

Til slutt er studien avgrenset til statisk analyse og manuell gjennomgang. Dynamisk testing, penetrasjonstesting og observasjon over flere utviklingsperioder kunne gitt ytterligere innsikt i sårbarheter, teknisk gjeld og langsiktige arkitektureffekter. SAIF-rammeverket er heller ikke eksternt validert av andre team, og adopsjonstrappens tidsrammer bør derfor forstås som anbefalinger basert på denne casen, ikke som erfaringsbasert bekreftede standarder.

6.4 Praktiske implikasjoner

Til tross for begrensningene gir studien flere praktiske implikasjoner for små utviklingsteam som vurderer å ta i bruk AI-kodeverktøy.

Sikkerhetsprompt er det viktigste enkelttiltaket. Med en økning på 98 prosent i sikkerhetsscore er dette det mest kostnadseffektive tiltaket et team kan innføre. Team som i dag bruker AI-verktøy uten sikkerhetsprompt bør innføre dette umiddelbart for all kode som berører autentisering, brukerdata eller hemmeligheter.

SAST alene er utilstrekkelig. Team som stoler på automatisert skanning som primært sikkerhetstiltak for AI-generert kode bør være oppmerksomme på at dette kun fanger en brøkdel av sårbarhetene. Manuell gjennomgang med en strukturert sjekkliste er uunnværlig.

Modellvalg bør være en bevisst beslutning. Ulike modeller har ulike styrker. Team som bruker én modell til alt går potensielt glipp av bedre resultater. Å investere tid i å kartlegge og teste tilgjengelige modeller betaler seg tilbake gjennom høyere kodekvalitet og bedre tilpasset output.

Sikkerhet trenger ikke bremse utviklingen. Funnene viser at de mest effektive sikkerhetstiltakene (sikkerhetsprompt og sikkerhetssjekkliste) krever minimal ekstra tid.

Gradvis innføring er mulig og anbefalt. Team trenger ikke implementere et komplett sikkerhetsrammeverk fra dag én. SAIF-rammeverkets adopsjonstrapp viser at selv det enkleste trinnet (sikkerhetsprompt) gir vesentlig forbedring. Teamet kan utvide prosessen etter hvert som kapasitet og erfaring øker.

7. Konklusjon

7.1 Svar på problemstillingen

Denne oppgaven har undersøkt hvordan små team i en bedrift kan integrere AI-akselerert utvikling i smidige arbeidsmetoder for å balansere sikkerhet, arkitektur og pålitelighet, samtidig som de fremmer innovasjon og parallelt samarbeid.

Studien adresserer dermed forskningsgapet identifisert i kapittel 2.11, hvor det ble påpekt at det finnes begrenset forskning på hvordan AI-verktøy påvirker produktivitet, kodekvalitet og sikkerhetspraksis i små utviklingsteam med begrenset sikkerhetskompetanse.

Gjennom et sikkerhetseksperiment med fire AI-modeller og erfaringer fra daglig bruk av AI-verktøy i utviklingen av ToSee-plattformen har studien avdekket flere sentrale funn.

For det første viser studien at AI-kodeverktøy gir vesentlig produktivitetsøkning i småteam, men at koden som genereres uten eksplisitte sikkerhetskrav inneholder systematiske sårbarheter. Hardkodede hemmeligheter og usikre cookie-konfigurasjoner forekom i 100 prosent av kodesampler generert med nøytral prompt. Disse funnene bekrefter og utdyper eksisterende litteratur om sikkerhetsrisikoen i AI-generert kode.

For det andre demonstrerer studien at bruk av sikkerhetsprompt er et ekstremt effektivt tiltak. Gjennomsnittlig sikkerhetsscore økte fra 5.875 til 11.625 av 12, en forbedring på 98 prosent. Dette er et tiltak som krever minimal ekstra innsats og kan innføres umiddelbart av ethvert team.

For det tredje avdekker studien at automatiserte sikkerhetsverktøy (SAST) alene er utilstrekkelige for AI-generert kode. Semgrep oppdaget sårbarheter i kun 2 av 16 kodesampler, mens manuell gjennomgang fant problemer i 12 av 16 samples, inkludert alle samples generert med nøytral prompt. Små team kan ikke stole på automatisert skanning som eneste sikkerhetstiltak.

For det fjerde viser erfaringene fra ToSee at ulike AI-modeller har komplementære styrker. Bevisst modellvalg per oppgavetype, der sikkerhetskritiske oppgaver tildeles modeller med høy baseline-sikkerhet, designoppgaver tildeles designsterke modeller, og backend-oppgaver tildeles funksjonelt orienterte modeller, gir bedre resultater enn å bruke én modell til alt.

Basert på disse funnene har oppgaven utviklet SAIF (Secure AI-Integrated Framework), et rammeverk med fem komponenter som små team kan ta i bruk for å integrere AI-verktøy i smidige arbeidsflyter på en sikker og produktiv måte. Rammeverkets beslutningsmodell,

sikkerhetsnivåsystem, sikkerhetssjekkliste, arbeidsflytmodell og adopsjonstrapp gir en praktisk og gjennomførbar tilnærming som balanserer produktivitet med sikkerhet.

Svaret på problemstillingen er dermed at små team kan integrere AI-akselerert utvikling i smidige arbeidsflyter ved å kombinere tre grunnprinsipper: bevisst modellvalg, sikkerhetsprompt som standardpraksis for sensitiv kode, og strukturert manuell gjennomgang med differensiert innsats basert på risiko. Denne kombinasjonen gjør det mulig å opprettholde høy utviklingstakt og innovasjonsevne uten å gå på akkord med sikkerhet, arkitektur eller pålitelighet.

7.2 Videre arbeid

Studien peker på flere områder for videre forskning.

Ekstern validering av SAIF. Rammeverket er utviklet basert på erfaringer fra ett prosjekt. En naturlig videreføring er å teste SAIF i andre småteam med ulike teknologistakker og erfaringsnivåer for å vurdere overførbarhet og identifisere nødvendige tilpasninger.

Større sikkerhetseksperiment. Eksperimentet med 16 kodesampler gir indikasjoner på mønstre, men et større eksperiment med flere modeller, oppgavetyper og kjøringer per kombinasjon ville styrke den statistiske kraften og generaliserbarheten.

Dynamisk sikkerhetstesting. Denne studien benyttet kun statisk analyse og manuell gjennomgang. Dynamisk testing (DAST) og penetrasjonstesting av kjørende applikasjoner kunne avdekke ytterligere sårbarheter som ikke fanges av SAST.

Langsiktige effekter. Studien dekker én utviklingsperiode. Longitudinelle studier ville gi innsikt i hvordan AI-akselerert utvikling påvirker teknisk gjeld og arkitekturkvalitet over tid.

Automatisering av SAIF-prosesser. Flere av rammeverkets komponenter kan automatiseres ytterligere — sikkerhetsnivåklassifisering kan integreres i oppgavestyringsverktøy, og sikkerhetssjekklisten kan implementeres som regler i CI/CD-pipelines.

Teamdynamikk og adopsjon. Videre forskning bør undersøke de menneskelige faktorene: hvordan påvirker AI-verktøy samarbeidsmønstre og kompetanseutvikling, og hva påvirker viljen til å adoptere sikkerhetsprosesser som SAIF?

Referanser

Anchore. (2025). DevSecOps best practices: A checklist to guide your team. https://anchore.com/devsecops/best-practices/

Apiiro. (2025, 4. september). 4x velocity, 10x vulnerabilities: AI coding assistants are shipping more risks. https://apiiro.com/blog/4x-velocity-10x-vulnerabilities-ai-coding-assistants-are-shipping-more-risks/

Atlassian. (u.å.). Agile software development for developers. Hentet 19. februar 2026 fra https://www.atlassian.com/agile/software-development

Becker, J., Rush, N., Barnes, E., & Rein, D. (2025). Measuring the impact of early-2025 AI on experienced open-source developer productivity. arXiv:2507.09089. https://arxiv.org/abs/2507.09089

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology*, *3*(2), 77–101. https://doi.org/10.1191/1478088706qp063oa

Bright Security. (2025, 25. mars). DevSecOps best practices – small changes for a big difference. https://www.brightsec.com/blog/devsecops-best-practices/

Cloud Security Alliance. (2025, 9. juli). Understanding security risks in AI-generated code. https://cloudsecurityalliance.org/blog/2025/07/09/understanding-security-risks-in-ai-generated-code

CodeRabbit. (2025). State of AI vs human code generation report. https://www.coderabbit.ai/blog/state-of-ai-vs-human-code-generation-report

Cui, Z., Demirer, M., Jaffe, S., Musolff, L., Peng, S., & Salz, T. (2025). The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. Management Science (forthcoming). https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4945566

CybSafe & National Cybersecurity Alliance. (2024). Oh, behave! The annual cybersecurity attitudes and behaviors report 2024-2025. https://www.cybsafe.com/whitepapers/oh-behave-the-annual-cybersecurity-attitudes-and-behaviors-report-24-25/

DORA Team, Google Cloud. (2024). Accelerate State of DevOps report 2024. https://dora.dev/research/2024/dora-report/

Farry, P. (2026, 3. februar). Working with code assistants: The skeleton architecture. InfoQ. https://www.infoq.com/articles/skeleton-architecture/

Fauscette, M. (2024, 31. desember). Agile product development with AI: Automating code, testing, and deployment. Arion Research LLC. https://www.arionresearch.com/blog/xnxespnb7d5hldlf9cu6knprbevwhq

GitHub. (2025). Octoverse 2025. https://github.blog/news-insights/octoverse/

GitClear. (2025). AI copilot code quality: 2025 data suggests 4x growth in code clones. https://www.gitclear.com/ai_assistant_code_quality_2025_research

GitLab. (u.å.). AI-driven code analysis: The new frontier in code security. Hentet 19. februar 2026 fra https://about.gitlab.com/topics/agentic-ai/ai-code-analysis/

IBM. (u.å.). What is DevSecOps? Hentet 19. februar 2026 fra https://www.ibm.com/think/topics/devsecops

JFrog. (2024, 7. februar). Analyzing common vulnerabilities introduced by code-generative AI. https://jfrog.com/blog/analyzing-common-vulnerabilities-introduced-by-code-generative-ai

Ji, J., Wun, J., Wu, M., & Gelles, R. (2024, november). Cybersecurity risks of AI-generated code (Issue Brief). Center for Security and Emerging Technology, Georgetown University. https://cset.georgetown.edu/wp-content/uploads/CSET-Cybersecurity-Risks-of-AI-Generated-Code.pdf

JetBrains. (2024). The State of Developer Ecosystem 2024. https://www.jetbrains.com/lp/devecosystem-2024/

JetBrains. (2025). The State of Developer Ecosystem 2025. https://blog.jetbrains.com/research/2025/10/state-of-developer-ecosystem-2025/

Kalva, R. (2024, 22. november). The evolution of DevSecOps with AI. Cloud Security Alliance. https://cloudsecurityalliance.org/blog/2024/11/22/the-evolution-of-devsecops-with-ai

Kashif, S. M., Liang, P., & Tahir, A. (2025). On developers' self-declaration of AI-generated code: An analysis of practices. ACM Transactions on Software Engineering and Methodology. https://doi.org/10.1145/3771937

Klemmer, J. H., Horstmann, S. A., Patnaik, N., Ludden, C., Burton, C., Jr., Powers, C., Massacci, F., Rahman, A., Votipka, D., Lipford, H. R., Rashid, A., Naiakshina, A., & Fahl, S. (2024). Using AI assistants in software development: A qualitative study on security practices and concerns. I CCS '24: Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (s. 2726–2740). https://doi.org/10.1145/3658644.3690283

Li, Z. S., Arony, N. N., Awon, A. M., Damian, D., & Xu, B. (2024). AI tool use and adoption in software development by individuals and organizations: A grounded theory study. arXiv:2406.17325. https://arxiv.org/abs/2406.17325

March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. Decision Support Systems, 15(4), 251–266. https://doi.org/10.1016/0167-9236(94)00041-2

Moreschini, S., Arvanitou, E.-M., Kanidou, E.-P., Nikolaidis, N., Su, R., Ampatzoglou, A., Chatzigeorgiou, A., & Lenarduzzi, V. (2026). The evolution of technical debt from DevOps to generative AI: A multivocal literature review. Journal of Systems and Software, 231, 112599. https://www.sciencedirect.com/science/article/pii/S0164121225002687?via%3Dihub

Oates, B. J., Griffiths, M., & McLean, R. (2026). *Researching Information Systems and Computing*. London: Sage.

OWASP. (2025). OWASP Top 10 for Large Language Model Applications. https://owasp.org/www-project-top-10-for-large-language-model-applications/

OX Security. (2025). Army of juniors: The AI code security crisis. https://www.ox.security/resource-category/whitepapers-and-reports/army-of-juniors/

Paradis, T., et al. (2024). How much does AI impact development speed? An enterprise-based randomized controlled trial. arXiv:2410.12944. https://arxiv.org/abs/2410.12944

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. 2022 IEEE Symposium on Security and Privacy (SP), 754–768. https://doi.org/10.1109/SP46214.2022.9833571

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv:2302.06590. https://arxiv.org/abs/2302.06590

Ramachandran, P. (2025, 17. november). Auto-generation is easy; auto-validation wins: Building reliability in AI-generated tests. StickyMinds. https://www.stickyminds.com/article/auto-generation-easy-auto-validation-wins-building-reliability-ai-generated-tests

RunSafe Security. (2025, 15. oktober). AI generated code and the next cyber crisis. https://runsafesecurity.com/blog/ai-generated-code-memory-protection/

Scarfone, K., Souppaya, M., & Dodson, D. (2022). Secure Software Development Framework (SSDF) Version 1.1: Recommendations for mitigating the risk of software vulnerabilities (NIST SP 800-218). National Institute of Standards and Technology. https://csrc.nist.gov/pubs/sp/800/218/final

Schreiber, M., & Tippe, P. (2025). Security vulnerabilities in AI-generated code: A large-scale analysis of public GitHub repositories. I ICICS 2025, Lecture Notes in Computer Science, vol. 16219. Springer. https://doi.org/10.1007/978-981-95-3537-8_9

Software AG. (2024). Shadow AI report. https://newscenter.softwareag.com/en/news-stories/press-releases/2024/1022-half-of-all-employees-use-shadow-ai.html

Sonar. (2025). OWASP LLM Top 10: How it applies to code generation. https://www.sonarsource.com/resources/library/owasp-llm-code-generation/

Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., Viswanath, B., & Jadliwala, M. (2025). We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. USENIX Security 2025. https://arxiv.org/abs/2406.10279

Stack Overflow. (2025). 2025 Developer Survey. https://survey.stackoverflow.co/2025/

Tambon, F., et al. (2025). Bugs in large language models generated code: An empirical study. Empirical Software Engineering, 30. https://arxiv.org/abs/2403.08937

Taware, V. (2025, 3. desember). How IT automation fuels lasting innovation. Dell Technologies Blog. https://www.dell.com/en-us/blog/how-it-automation-fuels-lasting-innovation

ThoughtWorks. (2025, mai). AI-first software engineering: Development, evolved. Perspectives (Edition 36). https://www.thoughtworks.com/en-us/perspectives/edition36-ai-first-software-engineering/article

Vaishnavi, V., & Kuechler, W. (2015). *Design Science Research Methods and Patterns* (2nd ed.). London: CRC Press.

Veracode. (2025, 30. juli). AI-generated code poses major security risks in nearly half of all development tasks [Pressemelding]. https://www.businesswire.com/news/home/20250730694951/en/

Verizon Business. (2025). 2025 Data Breach Investigations Report. https://www.verizon.com/business/resources/reports/dbir/

Walsham, G. (1995). Interpretive case studies in IS research: Nature and method. European Journal of Information Systems, 4(2), 74–81. https://doi.org/10.1057/ejis.1995.9

Wiz. (2026, 13. februar). 8 essential DevSecOps best practices. https://www.wiz.io/academy/application-security/devsecops-best-practices

Writer & Workplace Intelligence. (2025). 2025 AI survey: Generative AI adoption in the enterprise. https://writer.com/blog/enterprise-ai-adoption-survey/

Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2023). Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. arXiv:2304.10778. https://arxiv.org/abs/2304.10778

Yin, R. K. (2018). *Case Study Research and Applications: Design and Methods* (6th ed.). London: Sage.