CODORQ: ogrodje za orkestracijo razvoja programske opreme z različnimi agentskimi orodji

Tomaž Kolmanič

WagaLabs d.o.o., Maribor, Slovenija
tomaz.kolmanic@wagalabs.com

Prispevek predstavlja CODORQ, lokalno-prvo ogrodje za orkestracijo razvoja programske opreme z različnimi agentskimi orodji. Izhaja iz ugotovitve, da pri sočasnem delu več agentov omejitev ni le zmogljivost modelov, temveč predvsem odsotnost trajnega, preverljivega koordinacijskega sloja. CODORQ prek lokalnega strežnika MCP povezuje seje orodij Codex CLI, Claude Code in Antigravity CLI z repozitoriji, ločenimi Git worktreeji, terminali ter skupnim modelom nalog, vlog, statusov, pregledov, rezultatov QA in spomina. Sistem podpira ločitev dolžnosti med izvedbo, pregledom in QA, spremlja identiteto in živost sej ter človeka obravnava kot vodjo izvedbe. Trajni zapisi omogočajo obnovo konteksta po strnitvi ali ponovnem zagonu brez odvisnosti od terminalskega prepisa. Članek opiše arhitekturo, večstopenjski trajni spomin, statuse nalog, operaterski spletni vmesnik in obrambne mehanizme za ponovne zagone, neodzivne seje, preobsežne odgovore MCP in sočasne spremembe. Demonstracijski primer razvoja manjše komponente uporabniškega vmesnika pokaže sled od zahteve prek dokaznega commita in neodvisnega pregleda do rezultata QA. Prispevek ne dokazuje povečanja produktivnosti, temveč prikazuje, kako je mogoče večagentsko delo narediti sledljivo, obnovljivo in operatersko vodljivo.

Ključne besede:
agentska orodja
orkestracija razvoja programske opreme
Model Context Protocol
večagentski razvoj programske opreme
zagotavljanje kakovosti

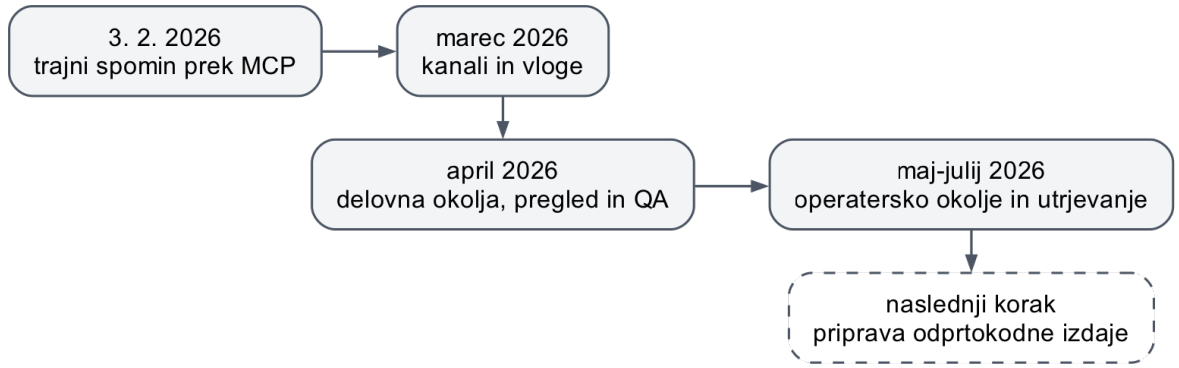
Univerzitetna založba Univerze v Mariboru

ISBN 978-961-299-188-3
DOI https://doi.org/10.18690/um.feri.7.2026.16

1 Uvod

Sodobni pomočnik AI za razvoj programske opreme lahko pregleda repozitorij, spremeni več datotek, zažene teste in pripravi razlago rešitve. Težave se pojavijo, ko hkrati poteka več takšnih sej. Ena pripravlja načrt, druga implementira rešitev, tretja pregleduje spremembo, četrta pa izvaja preverjanje kakovosti. Vsaka ima svoj kontekstni prostor in življenjski cikel. Terminal se lahko zapre, seja se lahko po strnitvi konteksta nadaljuje z novo tehnično identiteto, sprememba pa lahko ostane v ločenem delovnem drevesu Git (worktree) brez jasno določenega lastnika. Če je edini zapis o stanju pogovor v terminalu, mora operater ročno ugotavljati, kdo dela na kateri nalogi in kateri rezultat je dovolj zanesljiv za naslednji korak.

Razvoj sistema CODORQ se ni začel kot poskus vzpostavitve večagentskega sistema za razvoj programske opreme. Začetna zamisel je bila precej ožja: kako ohraniti uporabne odločitve, preference, pridobljene izkušnje in predaje tudi takrat, ko se pogovorno okno zapre ali delo prevzame druga seja. Prva različica sistema je zato predstavljala sloj trajnega spomina, dostopen prek protokola MCP. Praktična uporaba je kmalu pokazala, da sam trajni spomin ne zadošča. Zapisana odločitev namreč ne pove, ali je naloga pripravljena za izvedbo, kdo jo lahko prevzame, ali je bila sprememba pregledana in ali so bila zahtevana preverjanja dejansko izvedena. Sloj trajnega spomina se je zato postopoma razvil v izvajalno okolje za upravljanje nalog, vlog, pregledov, preverjanja kakovosti, terminalskih sej, usmerjanja ter operaterskega nadzora. Ta razvoj je mogoče strniti v prehod **od spomina k orkestraciji**. Slika 1 prikazuje ključne stopnje tega razvoja.



Slika 1: Strnjena časovnica razvoja CODORQ-a od trajnega spomina do orkestracijskega okolja. Zadnji korak označuje načrt, ne že izvedene javne objave.

Osrednja teza prispevka je, da zanesljivost večagentskega razvoja ni odvisna zgolj od zmogljivosti jezikovnega modela, temveč od celotnega delovnega sistema. Ta mora zagotavljati trajno stanje, jasno opredeljene prehode, izolirane spremembe, preverljive dokaze in odločitve človeškega vodje izvedbe. CODORQ to uresničuje kot lokalno-prvo koordinacijsko okolje, zgrajeno okoli obstoječih agentskih orodij. Modelu ne predpisuje načina razmišljanja niti ne nadomešča njegovega uporabniškega vmesnika. Namesto tega vsaki konkretni seji zagotavlja enoten nabor orodij za delo z delovnimi okolji, nalogami, trajnim spominom, pregledi, rezultati preverjanja kakovosti in kratkimi usmerjevalnimi sporočili.

Prispevek zasleduje štiri cilje. Prvič, predstavi arhitekturo lokalnega strežnika MCP, ki povezuje heterogena agentska orodja, repozitorije, terminale in trajno shrambo. Drugič, opiše delovni model, v katerem naloge, ugotovitve pregleda, rezultati preverjanja kakovosti in zapisi spomina predstavljajo samostojne poizvedljive entitete trajnega modela stanja. Tretjič, pojasni operativne varovalke, kot so ločitev dolžnosti, stabilizacija identitete, večstopenjsko zaznavanje neodzivnosti ter uporaba Git worktreejev. Četrtič, na dejanskem toku spremembe uporabniškega vmesnika prikaže pot od izvedbe prek pregleda do QA ter možnost naknadne rekonstrukcije dokazov.

Članek ne dokazuje, da CODORQ razvoj pospeši za določen odstotek ali da izboljša kakovost v primerjavi z drugimi ogrodji. Za takšne sklepe bi bilo treba vnaprej določiti primerjalno metodologijo, izhodiščne pogoje in analizirati več razvojnih projektov. Predstavljeni kazalniki zato predstavljajo opisna dejstva enega razvojnega primera. Njihov namen ni merjenje splošne produktivnosti, temveč prikaz sledljivosti in izvedljivosti predlaganega procesa.

2 Ozadje in sorodna dela

Orodja agentov AI za razvoj programske opreme danes pokrivajo več različnih ravni avtomatizacije. Aider je pomočnik AI za programiranje v paru [1]. Codex CLI [2], Claude Code [3] in podobni odjemalci gredo korak dlje, saj lahko samostojno pregledujejo repozitorij, uporabljajo razvojna orodja in izvajajo ukaze. Takšna orodja so učinkovita znotraj posamezne seje, ne rešujejo pa vprašanj koordinacije med več sočasnimi sejami, kot so določanje lastništva nalog, sledljivost dela in predaja rezultatov.

Raziskovalna večagentska ogrodja se osredotočajo na drugačen vidik problema. ChatDev uporablja specializirane agente in komunikacijski tok za posamezne faze razvoja programske opreme, od načrtovanja do testiranja [4]. MetaGPT uvaja vloge in standardizirane operativne postopke kot mehanizem za strukturiranje sodelovanja med agenti [5]. SWE-agent raziskuje povezavo med jezikovnim modelom in računalniškim okoljem pri reševanju nalog programskega inženirstva [6]. Ta dela pomembno prispevajo k razumevanju načrtovanja in sodelovanja agentov, vendar obravnavajo predvsem vedenje večagentskega sistema. Prispevek CODORQ-a se osredotoča na drugo raven – trajno koordinacijo obstoječih terminalskih orodij agentov AI.

Najbližje primerjalne rešitve predstavljajo ogrodja za orkestracijo agentskih sistemov. AutoGen je uveljavljeno ogrodje za gradnjo aplikacij iz več sodelujočih agentov; predstavljeno je bilo na konferenci COLM 2024 [7]. Ob nastanku tega članka je projekt v vzdrževalnem načinu; Microsoft za nove projekte priporoča naslednika Microsoft Agent Framework [13], [14]. Podobno LangGraph in CrewAI ponujata programske abstrakcije za opis grafov oziroma skupin agentskih izvajanj. CODORQ ni zasnovan kot novo programsko ogrodje ali SDK za definiranje vedenja agentov. Njegov namen je vzpostaviti skupni lokalni koordinacijski sloj nad heterogenimi CLI-odjemalci različnih ponudnikov. Ta sloj zagotavlja trajno stanje, obnovljivo identiteto agentskih sej, jasno ločitev dolžnosti (angl. *separation of duties*, SoD), spremljanje živosti ter vključuje človeka kot vodjo izvedbe. Za delovanje sistema zato ni bistveno, ali posamezno orodje temelji na namenskem agentskem ogrodju ali predstavlja zgolj terminalski odjemalec. Ključno je, da je seja registrirana, ima določeno vlogo in uporablja skupni trajni delovni model.

Lokalno-prva zasnova CODORQ-a sledi načelu, da mora biti primarna kopija uporabnikovega delovnega stanja pod uporabnikovim nadzorom ter uporabna tudi brez odvisnosti od osrednje aplikacijske storitve [8]. V tem kontekstu izraz *lokalno-prvo* ne pomeni, da mora biti tudi jezikovni model izveden lokalno ali da sistem ne uporablja omrežja. Pomeni predvsem, da koordinacijski sloj, trajni zapisi, povezave s terminali in operaterski vmesnik delujejo v lokalnem okolju. Sam klic jezikovnega modela je lahko še vedno izveden prek zunanje storitve, odvisno od izbranega agentskega orodja.

Model Context Protocol (MCP) je bil v zborniku OTS 2025 že predstavljen v članku Janija Šumaka [9], zato ta prispevek ne ponavlja podrobnega opisa protokola. Za arhitekturo CODORQ-a sta bistveni dve lastnosti uradne specifikacije: standardizirana komunikacija med odjemalcem in strežnikom ter jasno opredeljena orodja z validiranimi vhodi [10]. MCP sam po sebi ne določa večagentskega procesa, temveč predstavlja komunikacijsko podlago. CODORQ ga uporablja kot skupni vstopni mehanizem v trajni delovni model, kar različnim odjemalcem omogoča izvajanje enakih operacij brez razvoja ločenih integracij za posamezne družine orodij.

Arhitekturna neodvisnost sistema ne pomeni, da so vse družine agentskih orodij v vsakem trenutku enako podprte. Njihove zmogljivosti in življenjski cikli se hitro spreminjajo. V povzetku so kot primeri navedeni Codex CLI, Claude Code in Antigravity CLI [11]. Googlovo orodje Antigravity CLI temelji na modelih Gemini. Ob nastanku tega članka so zato neposredno podprta orodja Codex CLI, Claude Code in Antigravity CLI, poteka pa razširitev

podpore na orodje OpenCode. Prispevek namenoma ne primerja kakovosti posameznih modelov niti ne predpostavlja trajne razpoložljivosti posameznega odjemalca.

3 Zahteve in načela zasnove

Zahteve sistema niso nastale zaradi želje po čim širšem naboru funkcionalnosti, temveč kot odgovor na ponavljajoče se probleme pri delu z več terminalskimi sejami. Tabela 1 povezuje opaženi praktični problem, iz njega izpeljano sistemsko zahtevo in mehanizem, s katerim jo CODORQ uresničuje.

Tabela 1: Od praktičnega problema do zahteve in mehanizma CODORQ-a

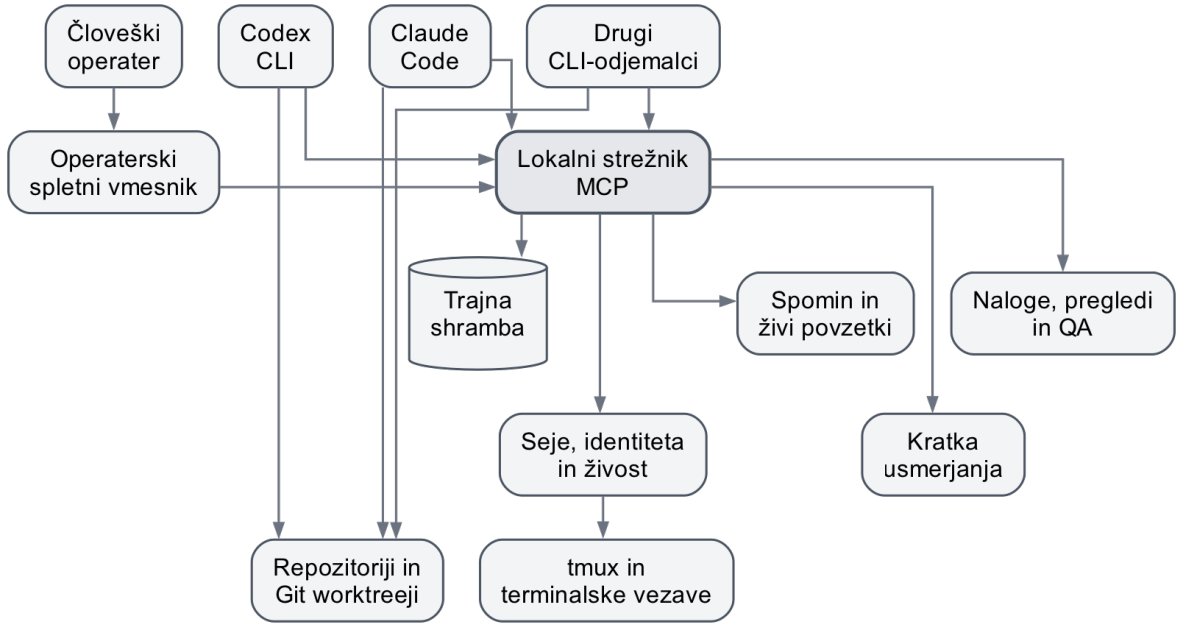
Problem	Zahteva	Mehanizem
Kontekst izgine ob zaprtju ali strnitvi seje	Trajna kontinuiteta	Spomin, kontrolne točke, živi povzetki, predaje in strukturirano stanje nalog
Več agentov si različno razlaga trenutno stanje	En poizvedljiv vir resnice	Naloge, ugotovitve pregleda, QA in seje v trajni shrambi
Sprememba nima jasnega lastnika ali vloge	Eksplicitna odgovornost	Vloge, dodelitve, zahtevane vloge, statusni prehodi in dogodki
Avtor sam potrdi svojo spremembo	Ločitev dolžnosti	Ločene izvedbene faze ter faze pregleda in QA z evidentirano sprostitvijo
Sočasni agenti spreminjajo isto delovno kopijo	Izolacija sprememb	Ločeni Git worktreeji in veja na nalogo
Terminal izgine, naloga pa ostane navidezno zasedena	Dokazljiva živost in okrevanje	Heartbeat, terminalska vezava, watchdog, reaper in obnovitev dodelitve
Odgovori MCP porabijo preveč kontekstnega prostora	Omejen in namenski kontekst	Kompaktni pogledi, zgornje meje seznamov in postopno razkrivanje podrobnosti
Operater ne ve, kje je potreben poseg	Nadzorni pogled in usmerjene akcije	Spletni vmesnik, kategorije pozornosti, zgodovina in kratka obvestila

Iz zahtev, predstavljenih v tabeli, izhajajo temeljna načela zasnove sistema. Prvo načelo je trajnost: komunikacijski kanal ali terminalska seja lahko posredujeta informacijo o dogodku, ne smeta pa predstavljati njenega edinega zapisa. Drugo načelo je eksplicitnost: vsaka naloga ima opredeljena merila sprejemljivosti in testni načrt, vsaka ugotovitev pregleda pa določen opis, resnost in stanje. Tretje načelo je najmanjša potrebna avtoriteta. Posamezna seja prejme le tista orodja in vlogo, ki jih potrebuje za izvedbo naloge, občutljivi prehodi pa vključujejo preverjanje identitete izvajalca. Četrto načelo je postopnost. Enostavne naloge ne zahtevajo večagentskega procesa, zahtevnejših izvedb pa ni mogoče izvesti mimo predvidenega procesa. Peto načelo je človeški nadzor. Avtomatizacija lahko predlaga nadaljnje korake, usmerja izvajanje ali pomaga pri obnovi dela, končna odločitev in odgovornost pa ostaneta pri operaterju.

4 Arhitektura in trajni model stanja

4.1. Koordinacijski in izvajalni sloj

CODORQ je zasnovan kot sistem dveh med seboj povezanih slojev. Koordinacijski sloj povezuje lokalni strežnik MCP, trajno shrambo, domenske storitve in spletni vmesnik. Izvajalni sloj sestavljajo agentski CLI-odjemalci, terminalske seje oziroma seje tmux, repozitoriji, Git worktreeji ter razvojna orodja, ki jih agenti kličejo. MCP predstavlja vstopno točko iz izvajalnega v koordinacijski sloj. Razmerje med slojema in povezanimi komponentami prikazuje slika 2.



Slika 2: Logične komponente CODORQ-a.

Tehnološki sklad sestavljajo Node.js 22, TypeScript in uradni SDK za MCP. Pot za lokalni razvoj in teste uporablja better-sqlite3, za resnejše večagentsko oziroma operatersko delo pa je podprta pot s PostgreSQL. Spletni del je zgrajen z Expressom in Reactom. Isti jezik tako pokriva CLI, strežnik MCP, spletni programski vmesnik (API) in odjemalski vmesnik, oba podatkovna adapterja pa ohranjata isti domenski model.

Merilni posnetek tik pred vključitvijo članka v repozitorij je obsegal 523 commitov od prvega commita 3. februarja 2026. Imenik src/ je vseboval 162 datotek TypeScript s 75.559 vrsticami kode, strežnik pa je objavljal 62 orodij MCP. To so opisni kazalniki obsega in zgodovine razvoja, ne merilo kakovosti ali produktivnosti. Vso programsko kodo sistema so napisali agenti AI pod nadzorom človeškega operaterja; od vzpostavitve osnovnih mehanizmov naprej je bil razvoj pretežno koordiniran prek CODORQ-a samega (pristop *dogfooding*).

Trajna shramba podpira različne tipe stanja: delovna okolja, registrirane agente in njihove seje, terminalske vezave, naloge in odvisnosti, ugotovitve pregleda, rezultate QA, dokumente spomina, kanalska sporočila in dogodke. Ne glede na izbrani podatkovni adapter je ključna enaka semantika: zapis je poizvedljiv tudi po koncu posamezne terminalske seje.

Vsak registrirani agent ima v delovnem okolju eno ali več vlog, na primer razvijalec, pregledovalec ali izvajalec QA. Vloga določa, katere naloge sme agent prevzeti in katere statusne prehode sme sprožiti. Posebna vloga je koordinacijski vodja: agent, ki cilje operaterja pretvarja v sledljive naloge, jih usmerja ustreznim vlogam in spremlja pretok dela do zaključka. Človeški operater zato praviloma komunicira z vodjo, ne z vsakim izvajalcem posebej.

Poleg trajnih zapisov CODORQ ponuja tudi kanale: sprotna sporočila med udeleženci delovnega okolja, ki se prenašajo prek lokalne vtičnice (angl. *Unix domain socket*).

Kanali imajo drugačno funkcijo kot trajna shramba. Uporabljajo se za kratko usmerjanje pozornosti, prebujanje ustreznega udeleženca in prenos sprotnih signalov v vmesnik. Uspešno poslano sporočilo še ne pomeni, da je naloga prevzeta ali končana. Avtoritativno stanje ostane v nalogi, ugotovitvi, rezultatu QA ali zapisu spomina.

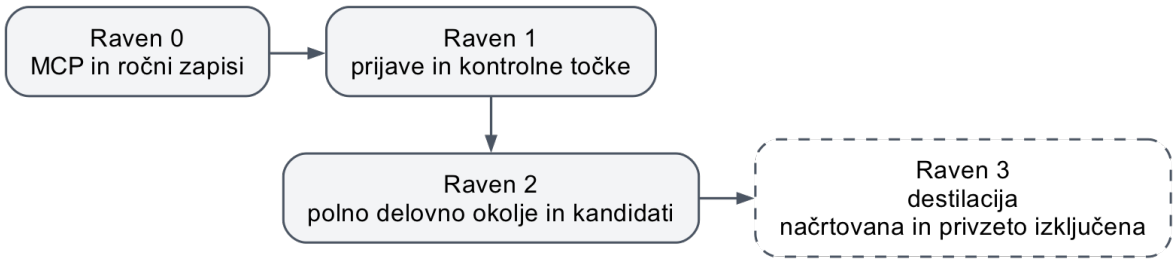
4.2. Trajni spomin kot prvovrstna komponenta

Trajni spomin je izvorna ideja sistema CODORQ in ostaja samostojen podsistem tudi potem, ko je projekt prerasel v orkestracijsko okolje. Njegov namen ni shraniti celotnega prepisa vsakega pogovora. Uporaben trajni spomin je izbran, strukturiran in umeščen v obseg. V njem se hranijo:

- pravila repozitorija in ponavljajoče se varnostne omejitve;
- preference operaterja ali ekipe;
- sprejete arhitekturne in izvedbene odločitve;
- naučene lekcije in ponavljajoči se problemi;
- predaje med sejami;
- kontrolne točke pred strnitvijo konteksta;
- strnjeni povzetki opravljenega dela in odprtih vprašanj.

Zapisi so razdeljeni v javne obsege, med drugim repo, global, preferences, insights, lessons, session in collaboration. Obseg zmanjša nevarnost, da bi lokalna odločitev enega repozitorija postala splošno pravilo za nepovezano delo. Iskanje po polnem besedilu deluje brez dodatne storitve. Semantično iskanje je mogoče vključiti z zunanjo storitvijo za izračun vektorskih vdelav (angl. *embeddings*) in povratnim polnjenjem obstoječih zapisov, vendar ni privzeta lastnost.

Način uporabe trajnega spomina je mogoče opisati s štirimi stopnjami. Na ravni 0 je na voljo strežnik MCP, agent pa orodja za trajni spomin kliče ročno. Raven 1 doda redne prijave, kontrolne točke in zajem ob koncu ali pred strnitvijo seje. Raven 2 spomin poveže s polnim delovnim okoljem: nalogami, živimi povzetki, pregledi, QA in pregledljivimi kandidati za prečiščeni spomin. Najvišja raven 3 je **destilacija (angl. *distillation*)**, pri kateri bi sistem iz zajetih dokazov pripravil sledljive predloge za trajnejše lekcije, odločitve ali nadomestitve zastarelih zapisov. Ta raven je načrtovana in delno izvedena, samodejni urnik pa je privzeto izključen. Zaporedje stopenj prikazuje slika 3.

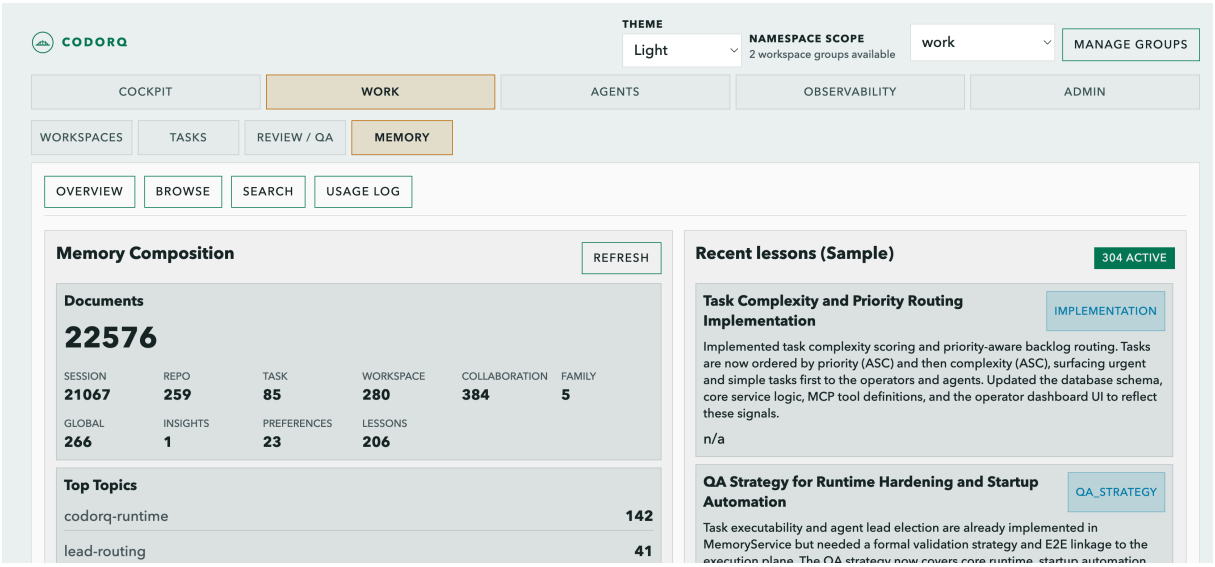


Slika 3: Stopnje uporabe trajnega spomina. Višja stopnja razširi prejšnjo.

Zapis spomina in izvršljivo stanje nista ista stvar. Lekcija »pri spremembi statusnega modela preveri tudi dokumentacijo« je primerna za spomin. Podatek, da določena naloga trenutno čaka QA, sodi v nalogo in evidenco QA. Živi povzetki, ugotovitve, rezultati QA in kontrolne točke so zato pri aktivnem delu pomembnejši od proste zbirke zapiskov. S tem se zmanjša možnost, da bi agent iz starega zapisa spomina sklepal o trenutnem lastništvu naloge.

CODORQ podpira tudi pregledljive kandidate za prečiščeni spomin. Predlog lekcije se lahko sprejme, zavrne ali nadomesti, preden postane del omejenega zagonskega konteksta. Zaznavanje protislovij in popolnoma samodejno razreševanje konfliktov v spominu ostajata odprta problema.

Slika 4 prikazuje operaterski povzetek sestave trajnega spomina in vzorec nedavnih lekcij. Števci predstavljajo stanje lokalne zbirke ob zajemu in niso merilo kakovosti ali stopnje ponovne uporabe zapisov.



Slika 4: Operaterski pregled sestave trajnega spomina in vzorca nedavnih lekcij. Posnetek je omejen na zgornji del pogleda, ki prikazuje vrste dokumentov, skupne teme in izbrane lekcije.

4.3. Kontinuiteta po strnitvi ali ponovnem zagonu

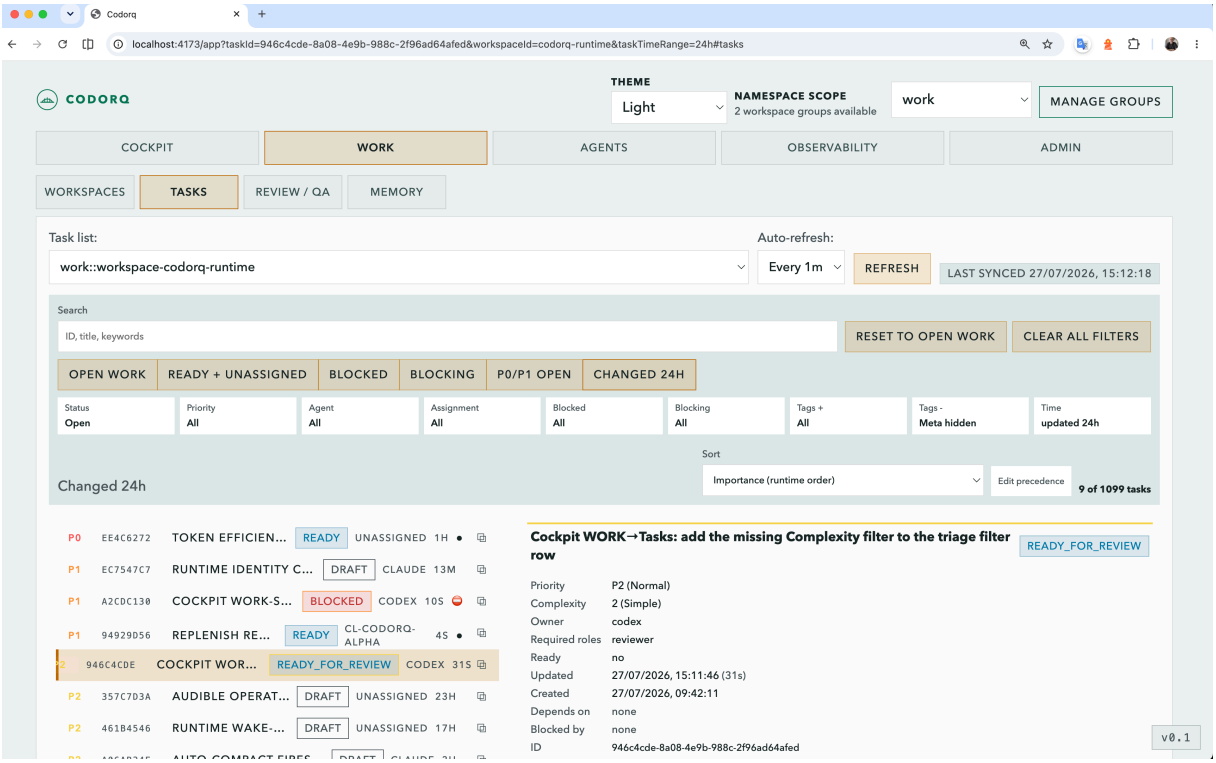
Pred strnitvijo obsežnega konteksta lahko seja zapiše kontrolno točko: aktivno nalogo, stanje, spremenjene datoteke, naslednje korake, odprta vprašanja in predaje. Ko se seja nadaljuje, ne potrebuje ponovitve celotnega pogovora. Naloži lahko kratek pogled workspace_my_work, podrobnosti aktivne naloge, odprte ugotovitve, zadnji rezultat QA in ustrezne poudarke trajnega spomina.

Primer obnovljenega delovnega stavka je lahko zelo kratek: »Nadaljujem popravek za stabilno identiteto. Implementacija je pregledana, QA pa je našel zastarel opis prioritet v dokumentaciji; popravi opis, zaženi ciljne teste in vrni nalogo v QA.« Tak zapis združuje trajno stanje več entitet, ne starega prepisa. To je operativna razlika med spominom kot arhivom in spominom kot delom obnovljivega procesa.

4.4. Repozitoriji, terminali in vmesnik

Vsaka izvedbena naloga lahko dobi svojo vejo in Git worktree [12]. S tem agenti ne spreminjajo istega delovnega imenika, pregledovalec pa lahko preveri točno določen commit. Terminalska vezava poveže registriranega agenta, izvajalno sejo in okno tmux. Trenutni dokaz iz okolja tmux ima pri obnovi prednost pred zastarelim shranjenim naslovom ali staro vezavo, vendar sistem pri dvomu ne sme izdelati nove identitete samo zato, da bi nadaljeval.

Operaterski spletni vmesnik na podlagi teh zapisov pripravi pregled delovnega okolja. Prikaže aktivno delo, čakalne vrste, blokade, zahteve za pregled, QA, agente in signale, pri katerih je potreben poseg. Slika 5 prikazuje dejanski lokalni pogled WORK → Tasks s filtri, seznamom nalog in inšpektorjem izbrane naloge. Posnetek prikazuje isti demonstracijski primer, ki je analiziran v 7. poglavju; osebni podatki, poverilnice in lokalne poti niso vključeni.



Slika 5: Operaterski pogled nalog v delovnem okolju. Posnetek je bil zajet iz lokalno delujočega sistema; prikazuje filtre, seznam nalog in izbrano nalogo za dodajanje filtra Complexity v stanju ready_for_review.

5 Izvedbeni tok in nadzor operaterja

5.1. Stopnje formalnosti

CODORQ ne predpisuje, da se vsaka zahteva obravnava kot večagentski projekt. Obseg koordinacije se prilagaja zahtevnosti dela. Tabela 2 prikazuje petstopenjski model formalnosti.

Tabela 2: Stopnje formalnosti izvedbe v CODORQ-u

Stopnja	Oblika dela	Prispevek CODORQ-a
0	Kratko vprašanje ali sporočilo v kanalu	Usmerjanje pozornosti brez izvršilne avtoritete
1	Posamezna sledljiva naloga	Lastnik ali vloga, status, merila sprejemljivosti in testni načrt
2	Krovna naloga	Živi povzetek, načrt, podnaloge in odobritev izvedbe
3	Večagentska izvedba	Ločene izvedbene steze, steze pregleda in QA, worktreeji ter nadzorovani prehodi
4	Več delovnih okolij ali ekip	Skupni pregled, obsegi spomina, dovoljenja, obnovitev in usmerjanje

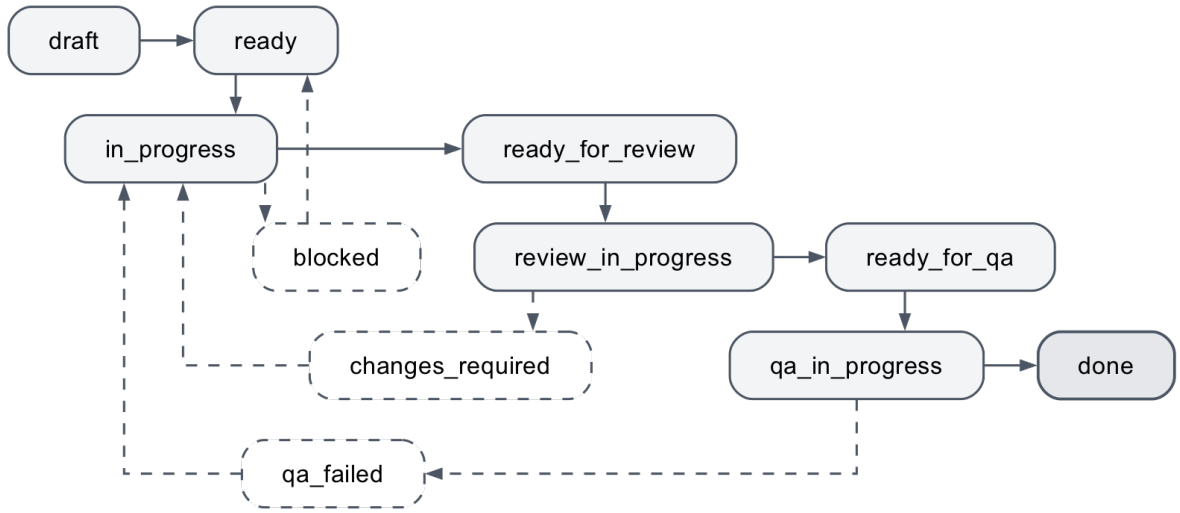
Če se kratka zahteva med izvedbo razširi, jo mora vodja pravočasno pretvoriti v sledljivo nalogo ali krovni delovni tok. S tem se zmanjša tveganje, da bi pomembne odločitve, odvisnosti ali odgovornosti ostale zapisane le v komunikacijskem kanalu.

5.2. Življenjski cikel naloge

Statusni model obsega dvanajst jasno opredeljenih stanj. Osnovni tok poteka od osnutka do zaključene naloge:

Osnovna pot: draft → ready → in_progress → ready_for_review → review_in_progress → ready_for_qa → qa_in_progress → done.

Poleg osnovnega toka model vključuje tudi stranske veje. Naloga lahko iz stanja in_progress preide v blocked in se po odpravi razloga vrne v ready. Pregledovalec lahko izbere stanje changes_required, po katerem avtor nadaljuje delo v in_progress. Podobno rezultat qa_failed vrne nalogo avtorju v nadaljnjo obravnavo. Dvanajsto, terminalno stanje je cancelled. Glavno pot in povratne prehode prikazuje slika 6.



Slika 6: Osnovni statusni tok naloge. Diagram prikazuje glavno pot ter vrnitev avtorju po zahtevanih spremembah ali neuspešnem QA.

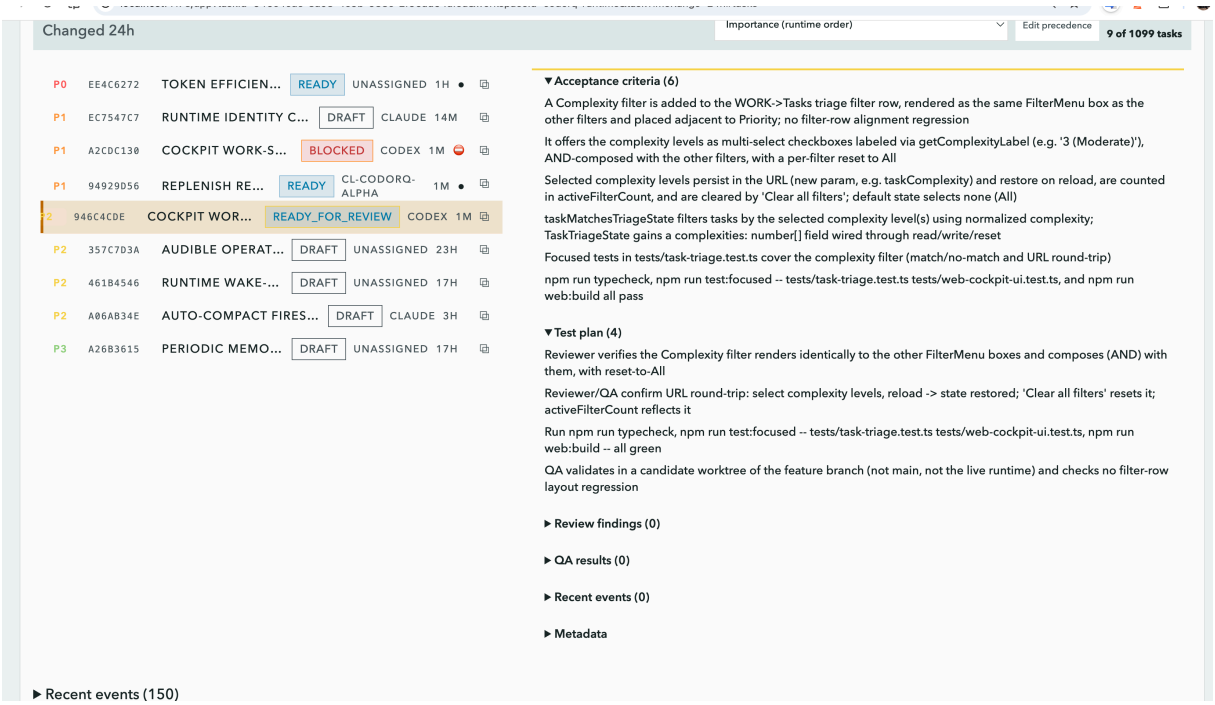
Dve pravili preprečujeta, da bi vmesnik prikazoval navidezno aktivnost. Prvič, aktivni statusi niso rezervacije. Stanja in_progress, review_in_progress in qa_in_progress pomenijo, da udeleženec nalogo dejansko izvaja v trenutnem delovnem obdobju. Če je ne izvaja nihče, mora naloga ostati v ustrezni pripravljeni čakalni vrsti. Drugič, stanje blocked privzeto ne pomeni zahteve za poseg operaterja, temveč operativno čakanje na odvisnost ali zunanji sistem. V operaterjev seznam se naloga uvrsti šele, ko vsebuje ekspliciten razlog čakanja na človeškega operaterja oziroma oznako awaiting_operator.

5.3. Od cilja do pregledanega rezultata

Večji cilj se najprej razčleni na živi načrt in podnaloge. Vsaka izvedbena naloga vsebuje opis, merila sprejemljivosti, testni načrt, zahtevano vlogo, oceno zahtevnosti in odvisnosti. Agent nalogo prevzame glede na svojo vlogo in trenutno razpoložljivost. Neposredna dodelitev je sicer mogoča, vendar ne nadomesti dokaza, da je seja živa in da je bila naloga dejansko prevzeta.

Po končani spremembi avtor zabeleži vejo, commit, spremenjene datoteke in rezultate preverjanj ter nalogo premakne v stanje ready_for_review. Pregledovalec ne odda zgolj prostega besedilnega mnenja, temveč strukturirane ugotovitve z določeno resnostjo, kategorijo, lokacijo, opisom, predlaganim popravkom in stanjem. Po odpravi ugotovitev naloga napreduje v QA. Rezultat QA vključuje izid, izvedena preverjanja, okolje, dokaze in opombe. Naloga se zaključi šele po uspešnem prehodu iz stanja qa_in_progress v done.

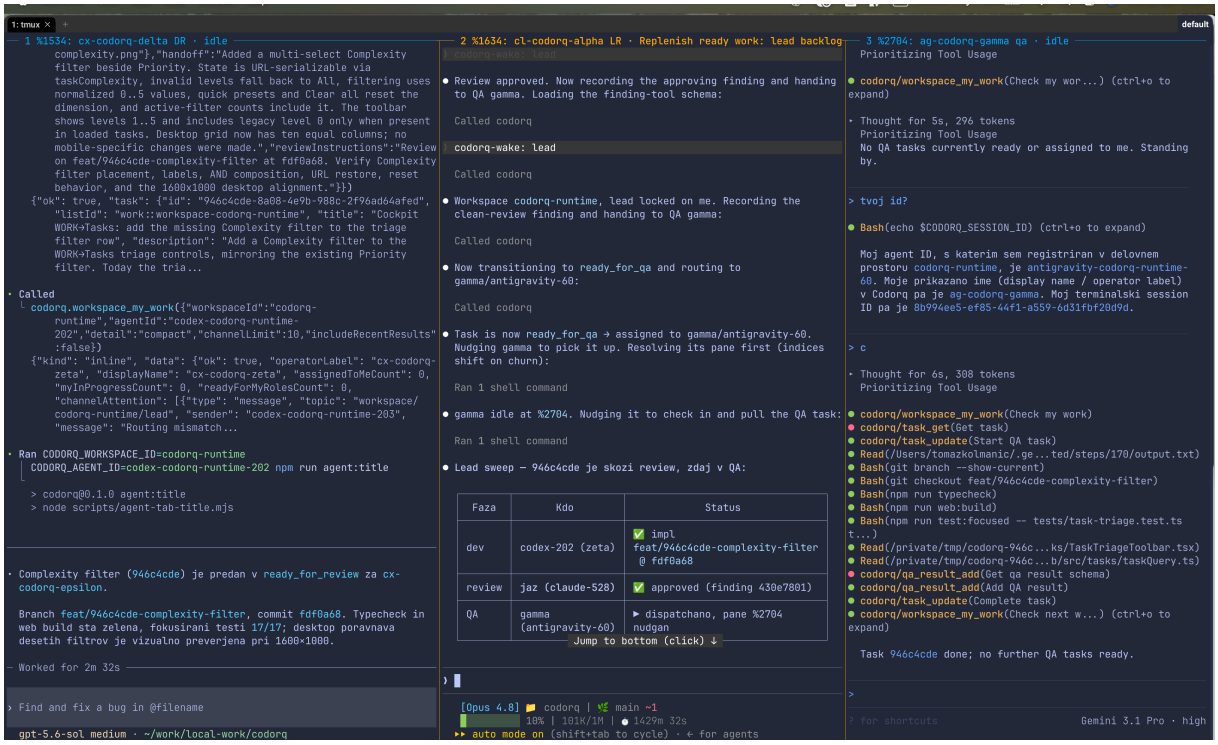
Slika 7 prikazuje podrobnosti iste naloge v stanju ready_for_review, z razprtimi merili sprejemljivosti in testnim načrtom. Seznam na levi ohrani kratek identifikator in status naloge, desni inšpektor pa zahteve, po katerih pregledovalec in izvajalec QA preverita rezultat.



Slika 7: Merila sprejemljivosti in testni načrt izbrane naloge v pogledu WORK → Tasks. Posnetek prikazuje nalogo v stanju ready_for_review pred neodvisnim pregledom in QA.

Ločitev dolžnosti pomeni, da avtor praviloma ne nastopa tudi kot končni pregledovalec ali izvajalec QA. V majhnih oziroma enoosebnih postavitvah takšna ločitev ni vedno izvedljiva. Sistem zato namesto tihega obhoda pravila omogoča eksplicitno sprostitev z ustreznim dokazom. Ključna je vidnost odstopanja: poznejši bralec mora biti sposoben ugotoviti, ali je rezultat preverila neodvisna seja.

Na sliki 8 je isti tok viden v ločenih terminalskih sejah izvajalca, vodje pregleda in izvajalca QA. Terminali so izvajalna in komunikacijska površina; avtoritativni dokaz o lastništvu, statusu, ugotovitvah in rezultatu QA ostaja v trajnem modelu CODORQ-a.



Slika 8: Izvedba iste naloge v ločenih terminalskih sejah za razvoj, pregled in QA. Vidni so interni klicni znaki agentov, ne pa osebni podatki; posnetek kaže transport navodil in dokazov med tremi ločenimi udeleženci.

5.4. Operater kot vodja izvedbe

Operater določa cilje, meje, prioritete in sprejemljivo raven tveganja. Vmesnik mu omogoča pregled nad aktivnimi delovnimi okolji, živostjo agentov, blokadami ter čakalnimi vrstami za pregled in QA ter zgodovino prehodov. Samodejno usmerjanje lahko poišče ustreznega agenta, ga opozori in zabeleži poskus dodelitve, ne more pa zagotoviti, da bo oddaljeni agent poziv vedno sprejel in nalogo pravilno izvedel.

Zato priprava toka pregleda in QA ni zadnji administrativni korak, temveč sestavni del načrta izvedbe. Že pred začetkom mora biti jasno, kdo sme pregledati spremembo, katera preverjanja so obvezna in kateri dokazi se pričakujejo. Operater lahko zadrži združitev, zahteva ponovni zagon okolja, spremeni razrez nalog ali omeji tvegano avtomatizacijo. Takšen nadzor je posebej pomemben pri spremembah identitete, terminalskih vezav in usmerjanja, pri katerih se napačni stranski učinki pogosto pokažejo šele v delujočem sistemu.

6 Praktični izzivi in obrambni mehanizmi

6.1. Identiteta seje po ponovnem zagonu

Agentsko orodje, proces, transport MCP, terminalska vezava in logični agent ne predstavljajo iste identitete. Ponovni zagon lahko ustvari nov identifikator transportne povezave, čeprav operater pričakuje nadaljevanje dela istega agenta. Če sistem brez dodatnega preverjanja ustvari novo agentsko instanco, lahko pride do podvojenih identitet, izgube neposrednih dodelitev in napačnega uveljavljanja ločitve dolžnosti.

CODORQ zato pri obnovi identitete uporablja več neodvisnih dokazov: eksplicitni parameter warmStartFrom, vezavo trenutne seje, trenutno okno tmux, klicni znak, družino orodja in stanje prejšnje instance. Živa vezava trenutnega okna ima prednost pred zastarelim zapisom. Zasedene vezave ni dovoljeno prevzeti zgolj zato, ker se nova seja predstavi z enakim imenom. Če razpoložljivi dokazi ne zadoščajo za zanesljivo odločitev, sistem zahteva poseg operaterja, namesto da bi ustvaril navidezno neprekinjenost dela.

6.2. Večstopenjski model živosti

Enoten časovni prag bi združil več različnih pojavov, zato ne bi zanesljivo odražal dejanskega stanja izvajanja. CODORQ ločeno spremlja živost terminalske vezave, aktivnost naloge in heartbeat registrirane agentske instance. Terminalska vezava lahko prehaja med stanji live, stale in transport_dead; watchdog zaznava nalogo, ki je formalno aktivna brez svežega dokaza dejavnosti, reaper pa po izteku registracije sproži obnovo dodelitve.

Vsi ti pragovi so nastavljivi in so odvisni od načina uporabe, terminalskega okolja ter pričakovane dolžine tihih operacij. Hitrejši signal terminalskega sloja je namenjen zgodnjemu opozorilu, počasnejši iztek agentske instance pa preprečuje prehitro odvzemanje dela ob krajših prekinitvah. Posamezna časovna vrednost zato ni del protokola CODORQ in je članek ne obravnava kot nespremenljivo lastnost sistema.

6.3. Varčevanje z žetoni (tokeni) in zmanjševanje šuma

Obsežna poizvedba lahko vrne več sto nalog, dolge opise, razporejevalne sledi in zgodovino dogodkov. Tak odgovor porabi pomemben del kontekstnega okna, še preden agent začne reševati svojo nalogo. CODORQ zato uporablja kompaktne poglede, vlogam prilagojene čakalne vrste, omejene sezname in postopno razkrivanje podrobnosti. Začetni klic vrne naslov, status, razlog vidnosti in najpomembnejše signale, podrobnosti posamezne naloge pa agent pridobi šele z namensko poizvedbo.

Takšen pristop neposredno zmanjšuje porabo vhodnih žetonov, vendar prispevek ne navaja odstotkovnega prihranka. Za kvantitativno oceno bi bilo treba vnaprej določiti nabor scenarijev, uporabljene modele, velikost kontekstnih oken, izhodiščni obseg podatkov in način štetja žetonov. Trenutni dokaz je zato arhitekturni: nepotrebna polja so iz začetnega pogleda odstranjena, dolgi seznami imajo določene zgornje meje, kontinuiteta pa se obnavlja iz kratke kontrolne točke namesto iz celotnega predhodnega pogovora.

6.4. Izolacija sprememb in vrstni red učinkov

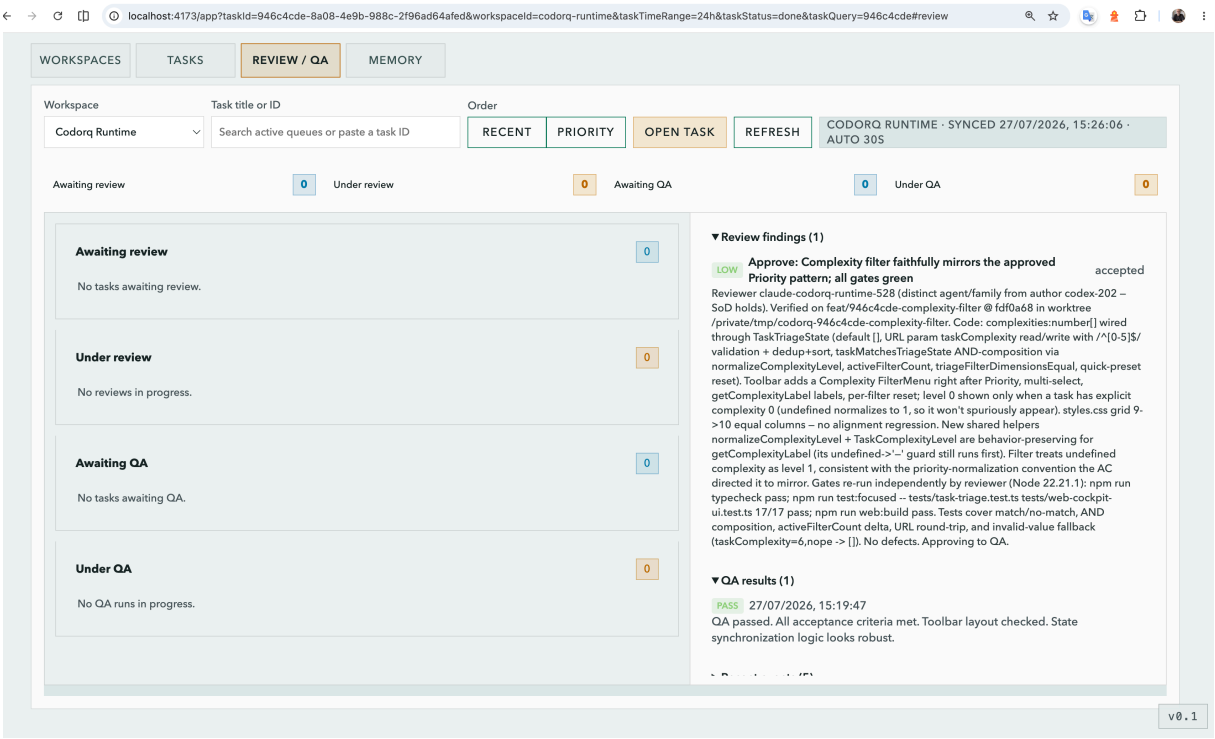
Ločeni Git worktreeji zmanjšajo verjetnost neposrednih konfliktov, ne odpravijo pa semantičnih neskladij. Dve ločeni rezini lahko na primer spremenita isto pogodbo prek različnih modulov. Zato ima vsaka naloga določen bazni commit, svojo vejo in dokazni commit, pregled pa mora preveriti razliko glede na pravilno osnovo.

Pri orkestracijskem jedru je pomembno tudi zaporedje stranskih učinkov. Posodobitev naloge lahko najprej preveri identiteto izvajalca in veljavnost statusnega prehoda, nato obnovi dodelitev, zapiše novo stanje, objavi dogodek in šele zatem sproži neblokirajoče prebujanje. Že dodan await ali drugačen vrstni red operacij lahko povzroči, da razporejevalnik zazna zastarelo stanje. Pri refaktoriranju zato ohranitev iste javne metode še ne zagotavlja enakega vedenja; preveriti je treba tudi asinhrone meje in zaporedje stranskih učinkov.

6.5. Pregled, QA in operaterski pogled

Pregledni tok mora biti določen pred začetkom izvedbe. Če vnaprej ni jasno, kdo je odgovoren za pregled in katera dokazila so obvezna, se naloge lahko kopičijo v napačni čakalni vrsti ali pa jih potrdi njihov avtor. Enako velja za QA: zgolj zelen testni izpis brez navedbe ukaza, okolja in obsega preverjanja ne predstavlja zadostnega dokaza.

Slika 9 prikazuje strukturirano ugotovitev pregleda in rezultat QA za izbrano nalogo. Prazne čakalne vrste na levi ne pomenijo izgube sledi: po zaključku toka ostanejo sprejeta ugotovitev, rezultat pass in povezava z nalogo poizvedljivi v desnem delu pogleda.



Slika 9: Trajna evidenca pregleda in QA v operaterskem pogledu. Posnetek prikazuje sprejeto ugotovitev pregleda ter ločen rezultat QA po zaključku naloge iz istega razvojnega toka.

Operaterski pogled mora zmanjševati informacijski šum, ne pa ga zgolj prenesti iz terminala v spletni vmesnik. Navadne notranje blokade zato niso samodejno obravnavane kot alarmi. Prednost imajo neuspešen QA, mrtva terminalska vezava, izrecna zahteva za poseg človeka, zastala naloga in manjkajoči pregled. Vmesnik je zasnovan kot funkcionalen lokalni nadzorni center, ni pa predstavljen kot dokončano produkcijsko orodje z zaključenim pregledom dostopnosti.

Tabela 3 povzema glavne probleme, obrambne mehanizme in preostale omejitve.

Tabela 3: Problemi, obrambni mehanizmi in preostale omejitve

Problem	Posledica	Obrambni mehanizem	Preostala omejitev
Nova seja prejme novo identiteto	Izguba dodelitve ali podvojena agentska instanca	warmStart, vezava seje in dokaz tmux	Različni CLI-odjemalci uporabljajo različne dogodke življenjskega cikla
Vezava ne pošilja heartbeatov	Poziv je usmerjen v mrtvi terminal	Pragovi live, stale in transport_dead	Terminal je lahko živ, aplikacija pa neodzivna
Naloga ostane v stanju in_progress brez dejanske dejavnosti	Navidezno aktivna čakalna vrsta	Nastavljiv watchdog in dokaz dejavnosti	Dolgotrajno tiho delo zahteva eksplicitен dokaz
Agent izgine	Delo ostane neposredno dodeljeno	TTL, reaper in obnovitev dodelitve	Iztek je namenoma zakasnjen in nastavljiv
Odgovor MCP je preobsežen	Manj kontekstnega prostora za reševanje naloge	Kompaktni pogledi in postopno razkrivanje	Prihranek je odvisen od scenarija in modela
Sočasne spremembe	Konflikt ali uporaba napačne osnove	Worktree na nalogo in bazni commit	Semantični konflikt še vedno zahteva pregled

Problem	Posledica	Obrambni mehanizem	Preostala omejitev
Avtor sam potrdi spremembo	Slabša neodvisnost preverjanja	Vloge za ločitev dolžnosti in eksplicitna sprostitev	Majhna ekipa včasih potrebuje dokumentiran nadomestni postopek
Preveč signalov v vmesniku	Operater spregleda pomemben dogodek	Kategorije pozornosti in eksplicitna operaterska dejanja	Prioritete in uporabniški vmesnik zahtevajo nadaljnje umerjanje

7 CODORQ razvija CODORQ: sledljivi tok spremembe od zahteve do QA

7.1. Kontekst in metoda

Med pripravo posnetkov za ta prispevek je operater v pogledu WORK → Tasks opazil, da je mogoče naloge filtrirati po statusu, prioriteti, agentu, dodelitvi, blokadah, oznakah in času, ne pa tudi po zahtevnosti. Koordinacijski vodja je opažanje pretvoril v samostojno nalogo z naslovom »Add the missing Complexity filter«, šestimi merili sprejemljivosti in štirimi skupinami preverjanj. Naloga je dobila prioriteto P2 in oceno zahtevnosti 2.

Primer je bil izbran zato, ker je nastal med pripravo članka in je bilo mogoče zajeti celoten tok v delujočem sistemu: triažo nalog, izvedbo v ločeni terminalski seji, strukturiran pregled, QA in končni zapis. Analiza temelji na trajnem zapisu naloge, metapodatkih evidence, commitu Git, eni strukturirani ugotovitvi pregleda in enem rezultatu QA. Terminalski izpis je uporabljen kot prikaz izvajanja in prenosa navodil, ne kot avtoritativni vir statusa.

Gre za eno majhno spremembo uporabniškega vmesnika, zato primer ni merilo splošne produktivnosti. Njegov namen je preveriti, ali je mogoče po zaključku rekonstruirati zahtevo, izvedbo, pregled in QA.

7.2. Izvedba in dokazni commit

Izvedbena seja družine Codex je nalogo prevzela v ločenem worktreeju na veji feat/946c4cde-complexity-filter. Zahtevana sprememba je obsegala stanje filtra complexities, serializacijo v parameter URL taskComplexity, normalizacijo vrednosti na lestvico 0–5, kompozicijo z drugimi filtri po pravilu AND, štetje aktivnega filtra in ponastavitev na vrednost »All«. V orodni vrstici je bil novi večizbirni meni postavljen neposredno za Priority, oznake pa uporablja iz obstoječe funkcije getComplexityLabel.

Dokazni commit fdf0a68 je spremenil šest datotek, dodal 74 in odstranil štiri vrstice. Poleg izvedbe je vseboval osredotočene teste za ujemanje in neujemanje, sestavljanje filtrov, povratni zapis stanja URL, neveljavne vrednosti, ponastavitev in števec aktivnih filtrov. Avtorjeva predaja je navedla naslednja uspešna preverjanja:

- npm run typecheck;
- 17 od 17 osredotočenih testov v task-triage.test.ts in web-cockpit-ui.test.ts;
- npm run web:build;
- preverjanje poravnave desetih filtrov pri ločljivosti 1600 × 1000.

Slika 7 prikazuje nalogo v stanju ready_for_review, slika 8 pa sočasni pogled v ločene seje izvajalca, vodje pregleda in izvajalca QA. Posnetka skupaj pokažeta razliko med triažo v operaterskem vmesniku in izvajanjem v terminalih.

7.3. Neodvisni pregled in QA

Pregled je opravila druga agentska instanca družine Claude. V strukturirani ugotovitvi kategorije design in nizke resnosti je preverila vezavo novega stanja skozi poizvedbeni model, validacijo parametra URL, sestavljanje po pravilu AND, ponastavitve ter postavitev menija za filtrom Priority. Pregledovalec je neodvisno ponovil preverjanje tipov, vseh 17 osredotočenih testov in spletno gradnjo. Napak ni našel, ugotovitev je bila sprejeta, naloga pa je napredovala v QA.

QA je izvedla ločena instanca orodja Antigravity CLI. Strukturirani rezultat ima izid pass in navaja šest preverjanj: enako obliko menija, sestavljanje filtrov, ohranitev izbire v URL, obnovo po ponovnem nalaganju, ponastavitev in števec aktivnih filtrov ter tri avtomatizirane ukaze za preverjanje tipov, teste in spletno gradnjo. Po tem je naloga prešla iz qa_in_progress v done. Slika 9 prikazuje sprejeto ugotovitev pregleda in ločen rezultat QA tudi potem, ko so aktivne čakalne vrste že prazne.

Ločitev dolžnosti je bila v končnem toku ohranjena: avtor, pregledovalec in izvajalec QA so bili tri različne agentske instance.

7.4. Vloga operaterja in meja sklepanja

Operater je zaznal manjkajoči filter, določil cilj spremembe in odobril uporabo tega toka za demonstracijski primer. Koordinacijski vodja je zahtevo pretvoril v sledljivo nalogo, določil merila sprejemljivosti ter delo usmeril skozi izvedbo, pregled in QA. Avtomatizacija je podprla čakalne vrste in statusne prehode, operater pa je ostal odgovoren za cilj, sprejemljivo tveganje in končno uporabo rezultata.

Iz enega primera ni mogoče sklepati, da takšen tok zmanjšuje čas razvoja ali število napak. Mogoče pa je preveriti, da so po zaključku na voljo zahteva, šest meril sprejemljivosti, veja, commit, rezultati avtorjevih preverjanj, neodvisna ugotovitev pregleda, ločen rezultat QA in končni status. Prav ta rekonstruirana sled je osrednji rezultat demonstracijskega primera.

8 Razprava in omejitve

CODORQ združuje dve medsebojno dopolnjujoči se obliki kontinuitete. **Kontinuiteta spomina** ohranja pravila, preference, odločitve in pridobljene izkušnje, medtem ko **izvajalna kontinuiteta** ohranja nalogo, odgovorno vlogo, status, ugotovitve pregleda, rezultate QA in dokazni commit. Prva brez druge ustvari dobro obveščenega agenta brez jasnega delovnega okvira, druga brez prve pa sicer ohrani stanje izvajanja, izgubi pa razloge za odločitve in možnost ponovne uporabe pridobljenega znanja. Prehod od sloja trajnega spomina k orkestraciji je zato mogoče razumeti kot povezovanje obeh oblik kontinuitete.

Lokalno-prvi pristop operaterju omogoča neposreden nadzor nad koordinacijskim stanjem, terminalskimi vezavami in delovanjem agentov. Hkrati pa prinaša tudi operativne zahteve. Lokalna podatkovna baza, procesi, tmux, dovoljenja, življenjski cikli agentov in odjemalci morajo biti pravilno nameščeni ter usklajeni. Sodelovanje prek MCP je sicer mogoče tudi brez terminalskega nadzora, vendar funkcije, kot so prebujanje agentov, zajem dokazov o terminalskih oknih in samodejno usmerjanje nalog, zahtevajo združljivo terminalsko okolje.

Pomembna omejitev sistema ostaja tudi varnostni model. Orodja MCP lahko spreminjajo naloge, zaganjajo delovne tokove in vplivajo na terminalske procese, zato lokalna izvedba sama po sebi ne zagotavlja varnosti. Potrebni so preverjanje identitete izvajalca, omejevanje dovoljenj, redakcija diagnostičnih podatkov in pravilo, da se skrivnosti ne zapisujejo v trajni spomin. CODORQ prav tako ne zagotavlja samodejne klasifikacije vseh občutljivih podatkov.

Predstavljeni demonstracijski primer ima več omejitev. Obravnava en sam sistem, ki ga razvija in uporablja isti avtor. Takšno okolje omogoča podroben vpogled v trajne zapise izvajanja, hkrati pa povečuje tveganje za pristranskost. Predstavljeni opisni kazalniki ne pomenijo primerjave z ročnim razvojnim procesom ali drugim večagentskim ogrodjem. Primer obravnava eno manjšo spremembo uporabniškega vmesnika, izvajalci pa so kljub ločenim agentskim instancam delovali v istem projektu pod vodstvom istega operaterja. Popolna sled naloge, ugotovitve pregleda in rezultata QA zato dokazuje možnost rekonstrukcije tega toka, ne pa splošne popolnosti dokazov ali vzročne izboljšave kakovosti in produktivnosti.

Druga pomembna omejitev je razvojna zrelost sistema. Semantično iskanje po spominu ostaja neobvezna komponenta, operaterski vmesnik pa zahteva nadaljnje preverjanje dostopnosti ter zmanjševanje informacijskega šuma. Tudi podpora različnim CLI-odjemalcem se spreminja skupaj z njihovimi razvojnimi cikli.

Razvoj trenutno poteka v zasebnem repozitoriju GitHub. Po koncu priprave na objavo je načrtovana odprtokodna izdaja, vendar ob nastanku članka še ni javnega repozitorija niti določenega datuma izdaje. Pred tem je treba poenostaviti namestitev, utrditi pot s PostgreSQL, izboljšati obravnavo robnih primerov identitete in usmerjanja ter zmanjšati šum in ovire dostopnosti v operaterskem vmesniku. Pregled kandidatov za trajni spomin je že uporabna eksplicitna pot; samodejna destilacija, konfliktne varovalke in nadzor semantičnega odmika ostajajo načrtovane zmožnosti, ne lastnosti trenutne izdaje.

Za prihodnje empirično vrednotenje bi bilo smiselno vnaprej določiti primerljiv nabor razvojnih nalog, vključiti izhodiščno ročno izvedbo ter sistematično meriti velikost odgovorov MCP, porabo žetonov, čas do pregleda, število ponovitev, delež odkritih regresij in uspešnost obnovitve prekinjenih sej. Takšna primerjalna študija presega obseg tega prispevka. Koristno bi bilo tudi merjenje operaterjeve kognitivne obremenitve, saj učinkovitosti večagentskega sistema ni mogoče ocenjevati zgolj na podlagi strojno zabeleženih dogodkov.

9 Zaključek

Prispevek predstavlja CODORQ kot lokalno-prvo ogrodje za orkestracijo razvoja programske opreme z različnimi orodji agentov AI. Osrednja ugotovitev prispevka je, da uspešno večagentsko delo ni odvisno zgolj od zmogljivosti posameznega jezikovnega modela, temveč predvsem od obstoja trajnega koordinacijskega sloja. Ta mora zagotavljati obnovljivo stanje, jasno razdelitev vlog, preverljive statusne prehode, ločitev dolžnosti in učinkovit operaterski nadzor.

CODORQ zato ne uvaja novega programskega ogrodja za razvoj agentov ali novega SDK-ja za njihovo vedenje. Namesto tega okoli heterogenih terminalskih odjemalcev vzpostavi skupen delovni model, ki vključuje trajni spomin, naloge, ugotovitve pregleda, rezultate QA, registracijo sej, upravljanje identitete, spremljanje živosti in usmerjanje izvajanja. Ključna lastnost sistema je, da lahko isti koordinacijski model uporabljajo različne družine CLI-orodij ne glede na njihovo notranjo implementacijo.

Predstavljena arhitektura temelji na lokalno-prvem pristopu. To pomeni, da koordinacijski sloj, trajni zapisi, povezave s terminali in operaterski pogled delujejo v uporabnikovem lokalnem okolju, medtem ko lahko izbrano agentsko orodje za izvajanje modelov še vedno uporablja oddaljene storitve. Model Context Protocol v tej zasnovi predstavlja standardiziran komunikacijski sloj, ne določa pa samega večagentskega procesa. CODORQ ga uporabi kot skupni vmesnik za izvajanje operacij nad trajnim delovnim modelom.

Demonstracijski primer filtra Complexity pokaže, da je mogoče zahtevo povezati z dokaznim commitom, avtorjevimi preverjanji, neodvisnim pregledom, rezultatom QA in končnim statusom. Tri različne agentske instance so pri tem opravile izvedbo, pregled in QA. Primer ne dokazuje večje produktivnosti ali kakovosti, dokazuje pa izvedljivost in sledljivost predlaganega modela.

Pomemben prispevek CODORQ-a je tudi vloga operaterja. Sistem človeka ne obravnava kot pasivnega nadzornika, temveč kot vodjo izvedbe, ki določa cilje, prioritete, sprejemljivo tveganje in ključne odločitve med celotnim življenjskim ciklom naloge. Večagentska avtomatizacija tako dopolnjuje človeško presojo in je ne nadomešča.

Predstavljena rešitev nastaja kot avtorjev samostojen projekt. Najbližji koraki so poenostavitev namestitve, utrjevanje identitete in usmerjanja ter izboljšanje operaterskega vmesnika. Obsežnejše empirično vrednotenje mora šele pokazati uporabnost pristopa v različnih razvojnih okoljih.

Literatura

[1] Aider, »Aider Documentation: AI Pair Programming in Your Terminal«. Dostopno na: https://aider.chat/docs/ (dostop 25. 7. 2026).

[2] OpenAI, »Codex CLI«, uradni repozitorij. Dostopno na: https://github.com/openai/codex (dostop 25. 7. 2026).

[3] Anthropic, »Claude Code Documentation«. Dostopno na: https://code.claude.com/docs/en/getting-started (dostop 25. 7. 2026).

[4] C. Qian idr., »ChatDev: Communicative Agents for Software Development«, v *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, str. 15174–15186, 2024. DOI: 10.18653/v1/2024.acl-long.810.

[5] S. Hong idr., »MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework«, *International Conference on Learning Representations (ICLR)*, 2024. Dostopno na: https://proceedings.iclr.cc/paper_files/paper/2024/hash/6507b115562bb0a305f1958ccc87355a-Abstract-Conference.html.

[6] J. Yang idr., »SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering«, *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. DOI: 10.52202/079017-1601.

[7] Q. Wu idr., »AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations«, *Conference on Language Modeling (COLM)*, 2024. Dostopno na: https://openreview.net/forum?id=BAakY1hNKS.

[8] M. Kleppmann, A. Wiggins, P. van Hardenberg in M. McGranaghan, »Local-first software: You own your data, in spite of the cloud«, v *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, str. 154–178, 2019. DOI: 10.1145/3359591.3359737.

[9] J. Šumak, »Protokol MCP – Kako povezati obstoječe aplikacije in agente«, v *Sodobne informacijske tehnologije in storitve: OTS 2025*, Univerza v Mariboru, str. 187–198, 2025. DOI: 10.18690/um.feri.7.2025.17.

[10] Model Context Protocol, »Architecture«, specifikacija 2025-11-25. Dostopno na: https://modelcontextprotocol.io/specification/2025-11-25/architecture (dostop 27. 7. 2026).

[11] Google, »Antigravity CLI Overview«. Dostopno na: https://www.antigravity.google/docs/cli-overview (dostop 27. 7. 2026).

[12] Git, »git-worktree Documentation«. Dostopno na: https://git-scm.com/docs/git-worktree (dostop 25. 7. 2026).

[13] Microsoft, »AutoGen«, uradni repozitorij. Dostopno na: https://github.com/microsoft/autogen (dostop 27. 7. 2026).

[14] Microsoft, »AutoGen to Microsoft Agent Framework Migration Guide«. Dostopno na: https://learn.microsoft.com/en-us/agent-framework/migration-guide/from-autogen/ (dostop 27. 7. 2026).