ChatGPT

Codex SaaS -projektin selvitys

1. Koodin ja skeeman nykytila

SQL-migraatiot ja näkymät: Projektissa on toteutettu useita tietokantamigraatioita Postgres-tietokantaan. Tärkeimmät migraatiot ovat:

- `migrations/0001_init.sql` – Alustaa tietokantataulut, tietotyypit ja triggerit. Tämä sisältää mm. *append-only* lokimekanismin sekä säännöt kuten "suunnitelma ennen ennustetta" (eli ennustetapahtumaa ei voi lisätä ennen kuin viimeisin suunnitelma on *READY_FOR_FORECAST* tai *LOCKED*) ja että mapping-rivejä saa muokata vain DRAFT-versiossa [1]. Näillä säännöillä varmistetaan baseline-lukituksen merkitys ja estetään historiatietojen muokkaus. Migraatio 0001 luo pohjan keskeisille tauluille (esim. budjetti/tavoitearvio, toteumat, work phase -rakenteet) ja liipaisimet, jotka toteuttavat mainitut liiketoimintasäännöt.

- `migrations/0002_views.sql` – Lisää tietokantaan näkymiä (*views*) ja apufunktioita raportointia varten [2]. Näkymät koostavat "nykytila"-raportteja, kuten työpaketin avainluvut ja projektin koontinäkymät. Esimerkiksi näkymät `v_report_work_phase_current` ja `v_report_project_current` laskevat kunkin työpaketin tai koko projektin budjetin, toteumat ja EV/CPI-arvot yhteen, jolloin käyttäjä saa yhdellä kyselyllä ajan tasalla olevat KPI-mittarit. Nämä näkymät hyödyntävät tauluihin tallennettuja tietoja (kuten baseline-arvot, prosenttivalmiusaste, toteutuneet kustannukset) ja laskentasääntöjä (ks. laskentasäännöt alla).

- `migrations/0003_jyda_snapshot_views.sql` – Päivittää näkymiä toteumatietojen snapshot-laskentaa varten [3]. Tällä *patch*-migraatiolla korjataan JYDA-järjestelmän tuomien toteumasnapshotien summaus: ilman korjausta samat toteumat saattaisivat summautua väärin eri ajankohtien näkymissä. Migraatio lisää tai korjaa siis näkymät, jotta toteumat ("Jyda-ajo") näkyvät oikein järjestelmässä. (Käyttötapaus: toteumatietojen tuonti ja raportointi, ks. alla toteuma.)

- **Myöhemmät migraatiot:** Lisäksi projektiin on lisätty SaaS-ominaisuuksia kuten monen organisaation tuki ja roolipohjainen käyttöoikeus (RBAC). Tätä varten on luotu omat migraatiot (Phase 19) tauluihin `organizations`, `users`, `roles` ym., sekä apufunktiot kuten `rbac_user_has_permission` ja tietoturvaan liittyvät *secure wrapperit* hyväksyntätoimintoihin [4] [5]. Terminologian monikielisyystuki (Phase 20) on toteutettu migraatioilla `0010_terminology_i18n.sql` ja `0011_fix_terminology_get_dictionary.sql`, joilla luotiin normalisoitu sanastotaulu ja haku**funktiot UI-teksteille eri kielillä [6]. Nämä varmistavat, että käsitteiden merkitys pysyy vakiona koodeina (esim. `BAC`, `EV`), mutta käyttöliittymässä näkyvät nimet voidaan määrittää organisaatio- ja kielikohtaisesti sanastossa.

Tietojen tuontiskriptit (import): Koodin osana on myös Python-pohjaisia työkaluja datajen tuomiseen Excel-lähteistä tietokantaan (kansiossa `tools/scripts/` [7]). Keskeiset skriptit ja niiden käyttötarkoitukset:

- `import_budget.py` – Lukee **tavoitearvion** Excel-tiedostosta (esim. *Tavoitearvio Kaarna Päivitetty.xlsx*) ja tuo koontitason budjettirivit tietokantaan `budget_lines`-tauluun [8].

Ensimmäisessä ajossa tällä tuotiin 206 riviä budjettidataa, joka koostui 109 ainutlaatuisesta nelinumeroisesta kustannuslitterasta (aggregoitua koodia) [9]. Skripti tunnistaa sarakkeet (esim. työvoima €, materiaali €, aliurakointi € jne.) ja tallentaa kullekin riville summat kustannuslajeittain sekä kokonaisSumman. Tämä muodostaa projektin baseline-budjetin pohjan.

- `import_budget_items.py` – Lukee tavoitearvion yksityiskohtaisella **nimiketasolla** ja tuo rivit `budget_items`-tauluun [10]. Tämä skripti täydentää edellä mainittua tuontia tuomalla kaikki yksittäiset nimikkeet (esim. tarkemmat erittelyt summista, joita aggregaattiriveissä on). Ensimmäisessä ajossa tuotiin 490 item-riviä, kun Excelissä rivejä oli ~777 – skripti suodattaa turhat/tyhjät rivit ja normalisoi tiedot niin, että vain oleelliset kustannusnimikkeet tallennetaan [11].

- `import_jyda.py` – Lukee **toteumatiedot** (todelliset kustannukset) JYDA-järjestelmästä tai vastaavasta laskentadata-Excelistä ja tuo ne `actual_cost_lines`-tauluun [12]. Skripti lisää toteumarivit ja päivittää samalla snapshot-näkymät, jotka edustavat eri ajankohtien kumulatiivisia toteumia. Tämän tuonnin myötä järjestelmään muodostuu mm. ajantasaiset toteumakustannukset työvaiheiden jäsenlitteroille sekä listaus “selvitettävistä” kustannuksista (ks. selvitettävät alla). Migraatio 0003 varmistaa, että nämä snapshot-näkymät laskevat summat oikein jokaiselle päivitykselle [3].

Keskeiset käyttötapaukset ja ominaisuudet: Edellä mainitut migraatiot ja skriptit palvelevat seuraavia ydinkäyttötapauksia Codex-projektissa:

- **Tavoitearvion käsittely (baseline/budjetti):** *Tavoitearvio* tarkoittaa projektin alkuperäistä budjettia. Excel-muotoinen tavoitearvio tuodaan tietokantaan `budget_lines`-tauluun, josta muodostetaan kullekin työpaketille sen oma budjettikooste. Työpaketin **baseline**-suunnitelma kootaan näistä budjettiriveistä: käyttäjä valitsee työpakettiin kuuluvat kustannusrivit ja voi tarvittaessa lisätä puuttuvan rivin, mikäli jokin tavoitearvion littera halutaan mukaan. Huomio: järjestelmän sääntöjen mukaan vain sellaiset rivit, jotka olivat olemassa tavoitearviossa, voidaan lisätä baselineen – uusia, täysin tunnistamattomia kustannuskoodeja ei baselineen sallita [13]. Mikäli jokin tavoitearvion rivi puuttui alun perin työpaketista, sen lisääminen baselineen edellyttää kaksiportaista hyväksyntää (ensin Työpäällikkö, sitten Tuotantojohtaja) [14]. Hyväksynnän jälkeen järjestelmä luo uuden baseline-version työpaketille, johon kyseinen rivi sisältyy [15]. Tämä varmistaa, että baseline pysyy hallittuna dokumenttina ja muutokset siihen ovat auditoituja.

- **Baseline-lukitus ja ennusteen seuranta:** *Baseline-lukitus* tarkoittaa, että työpaketin suunnitelma on virallisesti hyväksytty ja lukittu vertailutasona. Tietokantatasolla baseline lukitaan merkitsemällä työvaiheen suunnitelmatapahtuma (*planning_event*) tilaan LOCKED. Lukitus aktivoi järjestelmässä KPI-laskennan tälle työpaketille: vain lukitulle baseline-työpaketille lasketaan EV, CPI ym. arvot näkyviin [16]. Ennen kuin baseline on lukittu, järjestelmä **ei salli ennuste- tai toteumatietojen kirjaamista** kyseiselle työpaketille (DB-sääntö "suunnitelma ennen ennustetta") – käytännössä triggeri estää uuden forecast-tapahtuman tallentamisen, jos lukittua (tai ennusteelle avattua) baselinea ei ole [1]. Kun baseline on lukittu (policy A), työpaketti siirtyy *TRACK*-tilaan käyttöliittymässä, jossa voidaan tehdä viikkopäivityksiä: kirjata prosentuaalinen edistymä (% valmiusaste) sekä mahdolliset “haamukulut” (Ghost) ja muistiinpanot viikon aikana [17]. Lukitus takaa, että kaikki seurantaraportit perustuvat kiinteään lähtötasoon.

- **Mapping ja kohdistus:** Järjestelmässä on toteutettu *mapping*-toiminnallisuus, joka liittää kustannusrakenteen rivit (litterat) työpaketteihin. Mapping on versionoitu ja ajallisesti jäljitettävä: baselinea luotaessa muodostuu DRAFT-mapping-versio, jota voi muokata, mutta kun se aktivoidaan (lukitaan osana baselinea), siitä tulee *ACTIVE* eikä sitä saa suoraan muuttaa [18]. Uudet muutokset mappingiin tehdään luomalla uusi versio (esim. jos baselineen lisätään myöhemmin rivejä hyväksynnällä, syntyy uusi baseline-versio ja mapping-versio). Mapping-tiedot tallennetaan tauluihin (esim. `mapping_versions` ja `mapping_lines`), ja näkymät auttavat raportoimaan esim. mitkä budjettirivit on katettu työpaketissa (*coverage*). Tämä varmistaa, että jokainen toteutunut euro voidaan jäljittää tiettyyn baseline-riviin tai todeta "selvitettäväksi" jos ei kuulu mihinkään (ks. seuraava kohta).

- **Toteumatietojen tuonti ja *selvitettävät*:** Projektin **toteumat** (actuals) tuodaan säännöllisesti järjestelmään JYDA-ajona. Tuonnin tuloksena syntyy `actual_cost_lines`-dataa, jossa on kunkin kustannuslitteran toteutuneet eurot eri päivämäärillä. Järjestelmä laskee automaattisesti *snapshot*-näkymät, joista nähdään viimeisin tieto kunkin litteran kumulatiivisesta toteumasta sekä aiemmat tilanteet. Integroitu *raportointi* nostaa esiin myös **selvitettävät kustannukset**: nämä ovat toteumarivejä, joita ei ole vielä yhdistetty mihinkään työpakettiin (eli kustannus on kirjautunut projektille, mutta sitä vastaavaa baseline-riviä/työpakettia ei löydy). Projektin koontinäkymässä `v_report_project_current` näytetään mm. summa *UNMAPPED_ACTUAL_TOTAL* – tämä on selvitettävien kustannusten kokonaismäärä [19]. Lisäksi on erilliset näkymät esim. `v_actuals_latest_snapshot_unmapped` ja `v_selvitettavat_actuals_by_littera`, jotka listaavat detaljitasolla nämä kohdistamattomat rivit [20]. Käyttäjän on tarkoitus käsitellä selvitettävät joko liittämällä ne johonkin olemassa olevaan työpakettiin tai luomalla tarvittaessa uusi työpaketti, jotta kaikki kustannukset saadaan katettua suunnitelmaan. Näin järjestelmä auttaa löytämään "löysät" kustannukset, jotka muuten jäisivät raporteissa irrallisiksi.

- **Raportointinäkymät:** Edellä mainitut **views**-näkymät kokoavat tiedot yhteen. `v_report_work_phase_current` näyttää yksittäisen työpaketin nykytilan: sen budjetin (BAC), toteuman (AC), haamukulut (Ghost Open), lasketun Earned Valuen (EV), AC*:n (Actual+Ghost) ja CPI-indeksin, sekä mahdolliset kustannusylitykset (CV) ja muita mittareita.* `v_report_project_current` *summailee lukemat projektitasolle kaikista työpaketeista, mutta* vain *lukituista baselineista – työpaketit, joita ei ole lukittu, eivät kontribuoi EV/CPI-lukuihin (koska niitä ei lasketa ennen baseline-lukitusta, policy A)* [21]*. Projektinäkymä esittää myös selvitettävien kokonaissumman erillisenä rivinään, jotta johdon huomio kiinnittyy kohdistamattomiin kustannuksiin. Lisäksi on näkymä* `v_report_project_main_group_current` *pääryhmätasolla, joka niputtaa kustannuskoodit päätason kategorioihin. Kaikki nämä näkymät on toteutettu SQL:llä migraatioissa ja ne noudattavat alla kuvattuja laskentasääntöjä. Ensitestissä ajetut* smoke test *-kyselyt vahvistivat, että näkymät toimivat: esimerkiksi jos joltakin työpaketilta puuttui percent_complete**-arvo, niin näkymässä EV ja CPI näkyivät tarkoituksella tyhjinä (NULL) kyseiselle paketille [22], mikä on juuri haluttu toiminta sääntöjen mukaan (EV/CPI vaativat valmiusasteen).

Laskentasäännöt ja KPI-mittarit: Projektille on määritelty tietyt laskentasäännöt (BAC, AC, EV, CPI jne.), jotka on *lukittu merkitykseltään* – eli niiden määritelmät ovat päätetty ja koodi käyttää näitä vakioina kauttaaltaan [23]. Käytännössä nämä säännöt on toteutettu tietokantatasolla näkymien laskennassa sekä tunnisteina terminologia-taulussa. Alla keskeiset mittarit ja niiden määrittelyt (kaavat/ehdot), sekä missä ne skeemassa näkyvät:

- **BAC** (*Budget at Completion*, *Baseline €*) – työpaketin lukitun baselinen budjetti euroina [24]. Tämä arvo lasketaan kullekin työpaketille summana sen baselineen kuuluvista budjettiriveistä. Ehto: BAC-arvo näytetään/lasketaan vain työpaketeille, joilla on lukittu baseline (muuten baselinea ei

ole hyväksytty, joten BAC ei ole vahvistettu) [24]. Skeemassa BAC on tallennettuna esim. `work_phase_baseline`-taulussa (baseline-versioon liittyvä summa) ja sitä käytetään näkymissä suoraan työpaketin budjettiarvona.

- **PCT** (*Percent Complete*, *Valmiusaste %*) – työpaketin viimeisin raportoitu valmiusaste prosentteina (0–100%). Tämä tieto tulee viikkopäivityksissä käyttöliittymästä käyttäjän syöttämänä, ja tallennetaan todennäköisesti `forecast_event_lines`-tauluun tai vastaavaan (jokaiselle työpaketin riville voidaan merkitä edistymä). Jos valmiusaste puuttuu (NULL), järjestelmä tulkitsee, ettei EV/CPI-arvoja pidä näyttää (koska EV:n laskenta perustuisi tuntemattomaan valmiuteen) [25]. Näin ollen näkymissä EV ja CPI jäävät tyhjäksi kunnes %complete on syötetty kyseiselle työpaketille [22].

- **EV** (*Earned Value €*, *Ansaittu arvo*) – lasketaan kaavalla `EV = BAC * (PCT / 100)`, pyöristettynä kahden desimaalin tarkkuuteen [26]. EV kuvastaa työn arvosta toteutunutta osuutta euroina: esim. jos työpaketin budjetti (BAC) on 100 000 € ja valmiusaste 25 %, EV = 25 000 €. Ehto: EV lasketaan vain, jos työpaketilla on lukittu baseline *ja* valmiusaste on annettu (muuten EV ei määrittele) [26]. Skeemassa EV ei ole erillisenä kenttänä tallessa, vaan se lasketaan näkymissä lennossa edellä mainitulla kaavalla joka kerta (tai mahdollisesti tallennetaan johonkin *forecast_event* yhteenvetoon, mutta todennäköisimmin vain näkymässä).

- **AC** (*Actual Cost €*, *Toteutunut kustannus*) – kyseisen työpaketin piirissä toteutunut euro-määrä (kirjanpidon mukaan) [27]. AC saadaan summana `actual_cost_lines`-taulusta niille riveille, jotka on mappätty ko. työpaketin litteroihin. Käytännössä toteumat tuodaan projektitasolla, ja sitten näkymät summaavat kullekin työpaketille sen vastuulla olevat kustannukset. Huomiona projektissa on, että toteumatiedot tulevat viiveellä (esim. laskujen ja palkkojen sykli), joten AC voi päivittyä viiveelläkin [27]. Skeemassa AC on siis johdettu tieto: se näkyy `v_report_work_phase_current` -näkymässä laskettuna arvona (sum(Actual_cost_lines) tietyin ehdoin).

- **GHOST_OPEN** (“haamukulu”) – euro-arvo niille kustannuksille, jotka työmaa on kirjannut tehdystä työstä, mutta jotka eivät vielä näy kirjanpidon toteumassa [28]. Eli esimerkiksi työ on tehty ja kustannus tiedossa, mutta lasku ei ole vielä tullut ulos – tämä voidaan kirjata järjestelmään “Ghost” tietona, jotta ennusteessa osataan ottaa huomioon tuleva kulu. Haamukulu syötetään viikkoraportoinnin yhteydessä käyttöliittymässä ja tallentuu tietokantaan (esim. osana forecast_eventia). Skeemassa Ghost-arvo on mukana näkymän laskennassa: se haetaan viimeisimmästä päivityksestä kyseiselle työpaketille. Jos Ghost-tietoja ei ole annettu, arvo on 0.

- **AC*** (*AC + Ghost*, joskus kutsuttu “Actual + Ghost”) – *laskettu tunnusluku, joka on toteuman ja haamukulun summa:* `AC_STAR = AC + GHOST_OPEN` *(pyöristettynä kahteen desimaaliin)* [29]. *Tämä edustaa ns.* toteutunut + ennakoitu *kokonaiskustannus. AC* lasketaan näkymässä lennosta lisäämällä em. komponentit. Ehto: AC* on aina laskettavissa, mutta sen arvo on tietenkin sama kuin AC jos Ghost-arvoa ei ole.

- **CPI** (*Cost Performance Index*, *kustannustehokkuusindeksi*) – mittari, joka lasketaan kaavalla `CPI = EV / AC_STAR` (pyöristettynä neljään desimaaliin) [30]. CPI kertoo suhdelukuna, miten tehokkaasti rahaa on käytetty suhteessa työn edistymään (CPI = 1.0 tarkoittaa budjetin mukaisesti, yli 1 tarkoittaa alitusta, alle 1 ylitystä). Ehto: CPI:n laskemiseksi vaaditaan, että EV on laskettavissa (eli ei NULL) ja että AC > *0* [30]. *Jos AC* on 0 (eli ei kustannuksia vielä), CPI:lle ei ole mielekästä arvoa, ja jos EV on NULL (puuttuu baseline tai PCT), CPI jää myös NULL. Skeemassa

CPI lasketaan näkymissä joka kerta näillä ehdoilla; kun työpaketin EV ja AC* päivittyvät, CPI päivittyy automaattisesti näkymässä.

- **Muut mittarit:** *Cost Variance (CV)* eli kustannuspoikkeama lasketaan EV:n ja AC:*n erotuksena (EV – AC*), ja sitä voidaan esittää myös muodossa *"overrun"* = AC – *EV tilanteesta riippuen* [31] *. Nämäkin näkyvät todennäköisesti työpaketin raportointinäkymässä sarakkeina tai ainakin johdettavissa luvuista. UNMAPPED_ACTUAL_TOTAL (ks. selvitettävät yllä) on erikoistapaus projektitason näkymässä: se esitetään erillisenä rivinä tai kenttänä, joka näyttää projektille kirjatun kustannusmäärän, jota ei ole vielä saatu liitetyksi mihinkään työpakettiin* [19] *. Tämän avulla projektin johdolla on näkyvyys "löysään rahaan" ja kannustin kohdistaa ne pikaisesti. Kaikki nämä laskentasäännöt on kirjattu myös projektin dokumentaatioon ja ne on huomioitu käyttöliittymässä terminologian kautta: jokaiselle koodille (BAC, EV, CPI jne.) on sanastossa yrityskohtaiset ja kielikohtaiset label-arvot. Esimerkiksi terminologia-taulusta haettava avain* `metric.bac` *tuotti suomenkieliselle organisaatiolle tekstin "Budjetti €" käyttöliittymässä* [32] *. Näin ollen skeemassa* `terminology_terms` *-taulu ja siihen liittyvät funktiot (* `terminology_get_dictionary` *) kytkeytyvät myös näihin laskentasääntöihin varmistamalla, että UI näyttää oikeat selitteet mittareille ilman, että ne on kovakoodattu sovellukseen* [33] .

2. Kuinka varmistaa, että käytössä on "uusin" koodi?

Jotta varmistetaan, että aina käytetään projektin **uusinta koodia**, suositellaan hyviä versionhallinta- ja käyttöönottokäytäntöjä. Ensinnäkin kaikki koodi – mukaan lukien SQL-migraatiot, skriptit ja dokumentaatio – tulisi olla **versionhallinnassa (Git)**. Käytännössä kehittäjät tekevät muutokset Git-repositorioon, ja esimerkiksi jokainen ominaisuus tai korjaus on omana commitinaan/haarana. Versionhallinta varmistaa, että tiimillä on yhteinen näkemys koodin tilasta ja että voidaan tarvittaessa palata aiempiin versioihin.

Fresh install -testit: On erittäin hyödyllistä säännöllisesti testata projektin asennus alusta loppuun puhtaalta pohjalta. Tämä tarkoittaa, että otetaan uusi ympäristö tai tyhjä tietokanta ja ajetaan kaikki migraatiot alusta alkaen oikeassa järjestyksessä. Näin varmistetaan, että dokumentoidut migraatiot riittävät luomaan toimivan skeeman tyhjästä, eikä koodissa ole piileviä riippuvuuksia, jotka olisivat jääneet huomaamatta. Projektin suunnitelmissakin on mainittu tarve "fresh install test" -ajolle ja migraatioiden järjestyksen tarkistamiselle [34] . Fresh install -testi kannattaa automatisoida esimerkiksi CI/CD-putkessa: jokaisen releasen tai merkittävän päivityksen yhteydessä pipeline pystyttää testitietokannan, ajaa migraatiot ja mahdollisesti esitäyttää sen perusaineistolla (kuten mallidata budjetille ja toteumille) varmistaakseen, että sekä skeema että logiikka toimivat odotetusti. Tämä prosessi auttaa havaitsemaan, jos jokin uusi migraatio unohtui commitoida, tai jos migraatioiden ajamisen järjestyksessä on ongelmia.

Migraatioiden oikea järjestys: Nykyisessä projektissa migraatiot on nimetty etunollin ja järjestysnumeroilla (0001, 0002, 0003, ...), mikä jo itsessään varmistaa aakkosjärjestyksen kautta oikean suoritusjärjestyksen [35] . Tämän lisäksi on hyvä käytäntö pitää kirjaa ajetuista migraatioista – esimerkiksi luoda tietokantaan erityinen migraatiohistoriataulu tai käyttää valmista migraatiotyökalua. Vaikka tässä projektissa migraatiot ajetaan ilmeisesti manuaalisesti (pgAdminilla tai skriptillä), voisi harkita kevyen migraatiohallintakirjaston käyttöä (kuten Pythonin Alembic tms.), joka kirjaa mitkä migraatiot on ajettu. Oleellista on, että **kaikki ympäristöt** (devaus, testi, tuotanto) päivitetään aina ajamalla puuttuvat migraatiot samassa järjestyksessä. Dokumentaatiossa on jo lueteltu, että uuden tietokannan luonnin jälkeen ajetaan 0001_init.sql ja sitten 0002_views.sql jne. tarkasti peräkkäin [35] – tästä linjasta ei tule poiketa. Mikäli versioita hallitaan Gitissä, on suositeltavaa tehdä jokaisesta

julkaisuversiosta *tagi* tai *release*, johon voidaan liittää tieto tietokannan versiosta (esim. “versio X vaatii migraatiot X asti ajettuna”).

Tietokannan ja koodin synkronointi: On tärkeää varmistaa, että tietokannan skeema vastaa aina sovelluskoodia. Yksi tapa tehdä tämä on yllä mainittu fresh install -testi, mutta tuotantoympäristössä tämä tarkoittaa käytännössä, että ennen uuden koodin käyttöönottoa ajetaan kaikki uudet migraatiot tuotantotietokantaan. Jos käytössä on erillinen testi/staging-ympäristö, siellä tulisi peilata toimenpiteet ensin: päivitä koodi, aja migraatiot, varmista että sekä skeema että data ovat kohdallaan (esim. testaa tärkeimmät näkymät kyselyillä tai UI:n kautta). **Schema dump vs. migraatiot:** Jotkut tiimit pitävät versionhallinnassa myös *schema dump* -tiedoston (eli koko tietokannan rakenteen yhdessä .sql-tiedostossa). Tämä voi olla hyödyllinen dokumentaationa ja nopeana tapana tarkistaa skeeman kokonaistilanne tai tehdä diff-menettelyllä erojen vertailua. Schema dump ei kuitenkaan korvaa migraatioita, vaan täydentää: kehittäjät voivat esimerkiksi päivittää schema dumpin aina, kun migraatioita on ajettu, jolloin se kuvastaa “viimeisimmän” skeeman tilaa. Tärkeää on varmistaa, että dump ei ala elää omaa elämäänsä irrallaan – se on vain johdettu artefakti migraatioista. Fresh install -testissä voidaan jopa generoida uusi schema dump ja verrata sitä versionhallinnassa olevaan dumpiin, jotta nähdään onko eroja.

Toinen keino on käyttää **testikantaa**: Pidetään jatkuvasti pystyssä tietokanta, johon kehitysmigraatiot ajetaan jokaisen sprintin tai ominaisuuden jälkeen, ja johon on ladattu testidataa (esim. pieni tavoitearvio, pari työpakettia, sample-toteumia). Tällä testikannalla ajetaan sovelluksen testejä tai vähintään manuaalisia kokeiluja. Näin kehittäjät havaitsevat nopeasti, jos uusi koodi ei toimi odotetusti skeeman kanssa. Esimerkiksi, jos jokin UI-komponentti odottaa tietokannasta saraketta joka onkin nimetty eri tavalla uudessa migraatiossa, testikanta/CI-testaus nostaa tämän esiin ennen tuotantoa.

Yhteenvetona: **versiohallinta + automaattiset testit** ovat avain. Git varmistaa koodin version näkyvyyden, ja automaattinen fresh install -migrointi testaa skeeman eheyden. Migraatioiden järjestys pysyy hallinnassa nimeämiskäytännön ansiosta, mutta tarvittaessa voidaan lisätä myös ohjelmallinen valvonta (migraatioloki taulussa). **Schema dump** on valinnainen lisäapu, joka helpottaa hahmottamaan kokonaisskeeman (hyödyllinen mm. uudelle kehittäjälle dokumentaatioon). **Testikanta** (tai useampia) on suositeltava jatkuvaan integraatioon – se varmistaa, että “uusin koodi” toimii yhteen “uusimman skeeman” kanssa ennen kuin muutokset hyväksytään lopullisesti.

3. Tekoälyavusteiset työkalut (esim. Codex) projektissa

Projektissa on jo hyödynnetty tekoälyavusteisia työkaluja (kuten OpenAI:n Codex/ChatGPT) eri kehitystehtävissä, ja niitä voidaan jatkossakin hyödyntää monipuolisesti. Raportoidusti Codex-tyyppisellä työskentelyllä on tuotettu mm. speksejä, migraatiopaketteja ja runbook-ohjeita projektin aikana [36]. Seuraavassa on eritelty, mihin osa-alueisiin AI-avusteinen koodigenerointi tai tarkistus voi tuoda hyötyä ja mitä rajoitteita on huomioitava:

- **Tietokantamigraatioiden luonti:** SQL-migraatioiden kirjoittaminen on strukturoitua työtä, jossa tekoäly voi auttaa esimerkiksi generoisemalla nopeasti alustavan CREATE TABLE -lauseen tai näkymän määrittelyn, kun kuvailet sille mitä haluat. Codexille voi syöttää esimerkiksi speksin tekstimuodossa (kuten vaikkapa “Tarvitaan taulu *organizations* jossa kentät X, Y, Z…”) ja työkalu tuottaa SQL-lauseet. Tämä on käytännöllistä: projektissa on todennäköisesti jo tehty näin migraatioita, koska Codexilla tuotettiin paketteja aiemmin [36]. On kuitenkin tärkeää **validoida** AI:n tuottama SQL huolellisesti – erityisesti varmistaen, että se noudattaa projektin omia sääntöjä. Esimerkiksi append-only -periaatteen (ei DELETE/UPDATE lauseita tapahtumatauluihin) tulee säilyä: tekoäly ei sitä automaattisesti tiedä, ellei sitä ohjeista. Siksi kehittäjän on

tarkistettava, ettei AI lisää esim. cascade delete -toimintoja tai muuta, mikä rikkoisi append-only-logiikkaa. Migraatiossa AI voi myös auttaa triggereiden tai funktioiden kirjoittamisessa (esim. tarkistamaan “suunnitelma ennen ennustetta” -sääntö triggereillä), kunhan liiketoimintasäännöt annetaan promptissa selkeästi.

- **UI-komponenttien ehdotus ja generointi:** Käyttöliittymäpuolella tekoälyä voidaan hyödyntää **rutiinikomponenttien** tai boilerplate-koodin generointiin. Nykyisessä Codex-projektissa, jossa on yksi ruutunäkymä (SETUP/TRACK-tilat työpaketille), voisi AI:lle antaa tietomallin kuvauksen ja pyytää generoimaan pohjan React/Vue-komponentille. Esimerkiksi kehittäjä voi kuvata, että “Tehdään taulukko, joka listaa kaikki työpaketit ja niiden BAC, EV, AC, CPI arvot; sarakkeiden selitteet haetaan sanastosta” – Codex voisi tuottaa esimerkkikoodin: HTML/JSX-taulukon oikeilla sarakeotsikoilla ja rivien läpikäynnillä. Tekoäly osaa hyödyntää tunnettuja komponenttikirjastoja tai plain HTML:ää rakenteen luontiin. Tämä voi nopeuttaa kehitystä huomattavasti, kunhan kehittäjä viimeistelee tuloksen vastaamaan tarkasti vaatimuksia. **Automaattinen UI-komponenttien rakennus skeemasta** on mahdollinen tiettyyn rajaan asti: esimerkiksi lomakekomponentin generointi tietokantataulun kenttien perusteella (CRUD-formi) onnistuu, jos skeemasta tai JSON-kuvauksesta kerrotaan AI:lle. Projektin tapauksessa voidaan periaatteessa syöttää Codexille JSON-esimerkki vaikka `v_report_work_phase_current` -näkymän tuloksesta (sisältäen kentät bac, ev, cpi jne.) ja pyytää sitä generoimaan käyttöliittymäkoodi, joka renderöi nämä tiedot taulukkoon. AI todennäköisesti osaa luoda perusrakenteen oikein. **On kuitenkin huomioitava**, ettei tekoäly tunne projektin erityisiä UI-ratkaisuja tai tyyliä ilman opastusta: generoidut komponentit pitää sovittaa projektin suunnittelumalleihin ja testata toimivuus. Myös monimutkaisemmat vuorovaikutukset (esim. baseline-lukituspyynnön 2-vaiheinen hyväksyntä UI:ssa) vaativat liiketoimintalogiikan ymmärrystä, jota AI:lla ei suoraan ole skeemasta. Niissä AI:ta voi silti hyödyntää avustamaan esimerkiksi käyttöliittymälogiikan jäsentelyssä (”mitä tapahtuu kun käyttäjä painaa Lukitse baseline -nappia”) ja vaikka tuottamaan pseudokoodia tai sekvenssikaavion tapaista kuvausta.

- **JSON-payloadien tarkistus ja generointi:** Mikäli projektiin kuuluu backend- ja frontend-kommunikointia JSON-muodossa (esim. REST API tai GraphQL), AI voi auttaa muodostamaan ja validoimaan näitä rakenteita. Kehittäjä voi syöttää Codexille vaikka Python-datamallin tai tietokantakaavan ja pyytää esimerkkipayloadia, tai päinvastoin antaa JSON-esimerkin ja kysyä, vastaako se määriteltyä skeemaa. Codex pystyy melko hyvin hahmottamaan rakenteet ja tuottamaan esimerkiksi testisyötteitä. Tämä on hyödyllistä testitapauksissa: voidaan generoida nopeasti useita erilaisia JSON-viestejä (oikeita ja tahallaan vääriä) palvelun sisääntulolle ja näin testata, miten backend reagoi. Samoin dokumentaatiossa tekoäly voi auttaa kuvaamaan API-payloadit selkokielellä tai kääntämään ne esim. Swagger/OpenAPI-muotoon. Toki, tietoturvan ja luottamuksellisuuden kannalta on harkittava, mitä dataa ulkopuoliselle AI-palvelulle uskaltaa syöttää – kenties käytetään vain rakenne, ei oikeita tunnisteita tai arvoja.

Rajoitukset ja tarkennukset (ettei Codex riko sääntöjä): Tekoälyä käytettäessä on tärkeää *ohjata* sitä projektin omilla **pelisäännöillä**. Tämä tarkoittaa, että kehittäjän on ymmärrettävä ja huomioitava projektin päätöslokit ja “policyt” promptissa tai ainakin tarkastuksessa. Esimerkiksi mainittu **policy A** (että EV/CPI lasketaan vain lukituille baselineille) on kriittinen liiketoimintasääntö [21]. Jos kehittäjä pyytää Codexia kirjoittamaan SQL-kyselyn KPI-mittareille, on hyvä erikseen mainita ehtona “laske EV ja CPI vain jos baseline_status = 'LOCKED'”. Ilman tätä ohjetta AI saattaisi tuottaa kyselyn, joka ei sisällä kyseistä ehtoa, jolloin se rikkoisi policy A:ta. Samoin *mapping*-säännöt ja *append-only* -periaate pitää tuoda esiin: kun pyytää AI:ta vaikkapa kirjoittamaan triggerin, on hyvä muistuttaa ettei päivityksiä ole sallittu tietyssä taulussa, tai että muutokset on tehtävä lisäämällä uusia rivejä versiotauluihin. **Tekoäly ei tiedä hiljaisia sopimuksia** ellei niitä kerrota – se hallitsee yleiset käytännöt, mutta projektikohtaiset “ei koskaan tee X” -säännöt on ihmisen vastuulla valvoa. Siksi on suositeltavaa ylläpitää selkeää

dokumentaatiota (kuten päätösloki, spec-määrittelyt) ja jopa syöttää niiden sisältöä AI:lle osana ohjeistusta. Esimerkiksi, voisi aloittaa promptin: “Meillä on seuraavat liiketoimintasäännöt: … [lista] … Kirjoita funktio joka tekee Y noudattaen näitä.” – näin Codexin tuotos todennäköisemmin kunnioittaa vaatimuksia.

Lisäksi on huomioitava, että AI saattaa tuottaa syntaktisesti oikeaa koodia, joka kuitenkin on semanttisesti väärin kontekstissa. Ihmisen pitää aina **reviewata AI-koodi**. Projektin tietoturvapolitiikat pitää myös huomioida: Codex saattaa esimerkiksi ehdottaa kovakoodattuja arvoja tai ohittaa käyttöoikeuscheckejä, ellei niitä erikseen mainita. Tässä projektissa on RBAC ja organisaatioeristys, joten AI:lle tulee muistuttaa että esim. kyselyissä on aina rajattava organisaatio-id:n perusteella data (multi-tenant eristys). Samoin UI-komponenttien generoinnissa on tärkeää, ettei AI kovakoodaa suomenkielisiä tekstejä tms., vaan käyttää sanastoa. Projektin päätöksissä on linjattu: **UI-tekstejä ei kovakoodata, vaan ne haetaan sanastosta** dynaamisesti [33]. Kun pyytää Codexia luomaan UI-elementin, tämän voi mainita (“käytä `terminology_get_dictionary` -funktiota tekstien hakemiseen avaimilla, älä kirjoita tekstiä suoraan”). Tekoälyä voi siis hyödyntää, mutta *kehittäjän vastuulla on varmistaa, että tuotettu koodi noudattaa kaikkia projektin sääntöjä ja käytäntöjä*.

Automatisoitu UI-komponenttien luonti skeemasta: Onko mahdollista, että nykyisessä Codex-projektissa UI rakentuisi automaattisesti skeeman perusteella? – **Osittain kyllä.** Koska projekti on datavetoisesti suunniteltu (selkeät taulut ja näkymät KPI-arvoille, budjeteille ym.), voidaan generointia hyödyntää UI-rakenteiden pohjalle. Esimerkiksi “One-screen” käyttöliittymän komponentit (asetukset vs. seuranta-näkymä) voitaisiin jossain määrin generoida: kun skeema kertoo, että työpaketilla on tietyt kentät (nimi, kuvaus, lead-littera) ja tietyt luvut (BAC, EV, AC, CPI, Ghost, PCT…), voidaan näistä johtaa lomakekenttiä ja raportointisarakkeita. Codexilta voi pyytää vaikka: “Tee React-komponentti, joka näyttää työpaketin tiedot ja listaa siihen kuuluvat kustannusrivit sekä lasketut KPI-arvot. Tässä JSON-esimerkki työpaketin datasta…”. AI tuottaa todennäköisesti kelvollisen alustavan koodin, jossa on taulukko kustannusriveistä ja yläosassa työpaketin perustiedot. Tämä automatisointi kuitenkin tuottaa **geneerisen toteutuksen** – se ei tunne käyttöliittymän toivottua ulkoasua tai tarkkaa käyttäjäpolkua. Siksi automaattigeneroinnissa saavutetaan ehkä 60-80% valmista koodia, jonka jälkeen ihmisen pitää räätälöidä loput. Hyöty on siinä, että kehittäjä saa nopeasti käsissään toimivan rungon, jota voi iteroida.

Nykyisessä projektissa voisi ajatella myös sellaista lähestymistapaa, että UI:n komponentit mallinnetaan esimerkiksi määrittelytiedostona (joko JSON tai jopa luonnollisella kielellä), ja AI:ta käytetään generaattorina tuottamaan koodi tuon mallin pohjalta. Tämä olisi eräänlainen *design-to-code* automatisointi. Käytännössä on kuitenkin varmistettava, että kaikki erityistapaukset (kuten painikkeiden toiminnallisuudet baseline-lukituksen pyytämiseksi, hyväksyntädialogit, “selvitettävien” käsittely-modalit jne.) lisätään jälkeenpäin.

Yhteenveto AI-avusta: Tekoälyavusteiset työkalut voivat tehostaa kehitystyötä monella rintamalla: ne kirjoittavat rutiinikoodia, ehdottavat rakenteita ja jopa dokumentoivat. Projektin kehityksessä Codex on jo auttanut migraatioiden teossa ja dokumentoinnissa [36], ja saman linjan voi jatkaa laajentaen UI- ja integraatiokerrokseen. Olennaista on jatkossakin käyttää AI:ta *avustajana*, ei korvaavana: ihmisen asiantuntemus tarvitaan validoimaan, ettei mikään projektille tärkeä sääntö tai laatuvaatimus rikkoudu. Kun tämä tasapaino pidetään, Codex-työkalusta voi tulla kehitystiimin tuottavuutta merkittävästi nostava apuri – se toteuttaa nopeasti kehittäjän aikeet koodiksi, ja kehittäjä varmistaa, että lopputulos täyttää **Codex SaaS -projektin** vaatimukset ja laadun. [21] [33]

---

[1] [2] [18] [35] README_DB.md.md
file://file-A9LmpQLNQ6ScyQQCCZ5XFB

3 README_JYDA_IMPORT.md.md
file://file-71cBY6qTFrRGv9tdqnXpR3

4 5 6 7 8 9 10 11 12 14 15 16 20 22 32 34 36 PROJECT_EVENT_LOG_SETUP.md
file://file-EWvKm3HbjgiQEJdnpB1i2b

13 21 33 MASTER_DECISIONS_SAAS_V1.md.txt
file://file-HDUujfGQnv8K2pyZegcSuK

17 UI_ONE_SCREEN_V1.md.txt
file://file-9gcAMHzUYVNiJy2K4r3o5D

19 23 24 25 26 27 28 29 30 31 SPEC_CALC_RULES_V1.md
file://file-7sxnXSJFv3F6WHpiwwWkK5