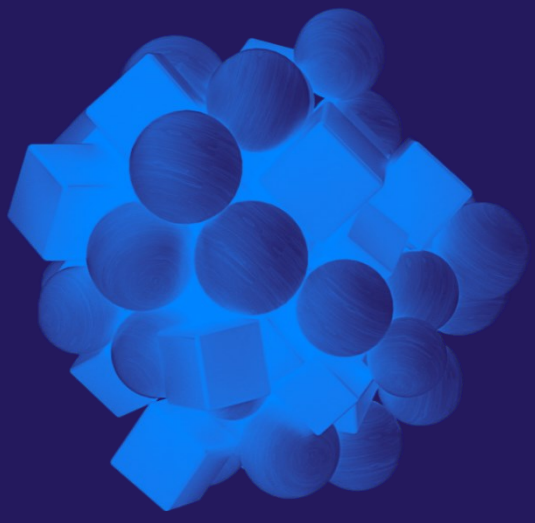
TECHBOOST

Case Finlandia-talo

-teknologiademoraportti



Yhteistyössä:

Finlandia-talo Oy

Elinkeino-, liikenne- ja ympäristökeskus
Työ- ja elinkeinoministeriö Arbets- och näringsministeriet
Tampereen ammattikorkeakoulu
Aalto-yliopisto Aalto-universitetet Aalto University
Metropolia
HAMK Hämeen ammatti-korkeakoulu
OAMK Oulun ammattikorkeakoulu
Tampereen yliopisto

Raportti Kirjoittajat: Fayez Bassalat, Ville Arffman

Mobiilirobotiikka Logistiikan Tukena – Case Finlandia-talo

Muotoilu, Hissi Integraatio, Logistiikka, Mekaniikka, Nostolaite, Automatisoiminen, REST API, Mobiilirobotiikka

Projektissa mukana olleet henkilöt:

Ville Arffman, Niko Hutri, Fayez Bassalat, Kalle Vänskä, Santeri Tirronen & Lauri Ojanen

Metropolia Ammattikorkeakoulu
23.12.2025
TECHBOOST-teknologiademoraportti

Sisällysluettelo

Lyhenteet

MiR:	Mobile Industrial Robots. Tanskalainen automaatiorobotteja valmistava yritys.
LiDAR:	Light detection and ranging. Mittaustyökalu, joka laskee anturin ja kohteen välisen etäisyyden.
CAD:	Computer Assisted Design. Nykyaikainen tapa tehdä tarkkoja 3d-malleja sekä valmistuskuvia kappaleista ja osista.
REST:	Representational State Transfer
API:	Application Programming Interfaces - sovellusrajapintoja
AMR:	Autonomous Mobile Robot. Autonominen mobiilirobotti
GUI:	Graphical User Interface. Graafinen Käyttöliittymä

1. Johdanto

Finlandia-talossa Helsingin Töölönlahdella on vuosikymmenten ajan järjestetty vuosittain satoja tapahtumia. Elämyksiä tuottavan talon toiminnassa on paljon liikkuvia osia, kalusteita ja teknisiä laitteita siirretään jatkuvasti tilasta toiseen. Työ on fyysisesti vaativaa ja tapahtuu usein tiukoilla aikatauluilla, öisin ja päivisin. Esimerkiksi muuttotiimi siirtää päivittäin jopa 1 500 tuolia juhlasalien ja varastojen välillä, useimmiten käsin nokkakärryillä. Tämä loi konkreettisen tarpeen logistiselle automaatioratkaisulle.

Vuonna 2022 Finlandia-talon yhteydessä toimivassa Pikku-Finlandiassa testattiin Metropolia-ammattikorkeakoulun Big Flash -hankkeessa Kaveri-robottia. Robotti kuljetti lisäosan avulla ruokalähetyksiä keittiön ja kahvion välillä, automatisoiden rutiinitehtävän, joka normaalisti sitoisi henkilöstöresursseja.

Tavoitteena oli vapauttaa työntekijöiden aikaa asiakaspalveluun ja testata robotiikan käytännön hyötyjä. Pilotointi koettiin onnistuneeksi niin työntekijöiden kuin asiakkaidenkin näkökulmasta ja tällaiselle kokeilulle haluttiin tehdä jatkumoa. Projekti tehtiin Metropolian Garage-konseptin Suvituuli ja Salama projekteina syksyllä 2022 ja keväällä 2023.

Kuva 1. Kaveri robotti Pikku Finlandiassa. (Big Flash Metropolia)

Uuden automaatio kokeilun kohteeksi valikoitui TECHBOOST-hankkeessa kehittää autonomista järjestelmää, joka pystyy noutamaan tuolipinoja kellarivarastosta ja toimittamaan ne hissillä juhlasaliin täysin itsenäisesti, myös yöllä ilman ihmiskontaktia.

Kuva 2. Finlandia talon tapahtumasali

Finlandia talo projekti alkoi syksyllä 2024, jolloin opiskelijaryhmä konseptoi MiR100-logistiikkarobotin perässä vedettävän vaunun tuolipinojen siirtämistä varten.

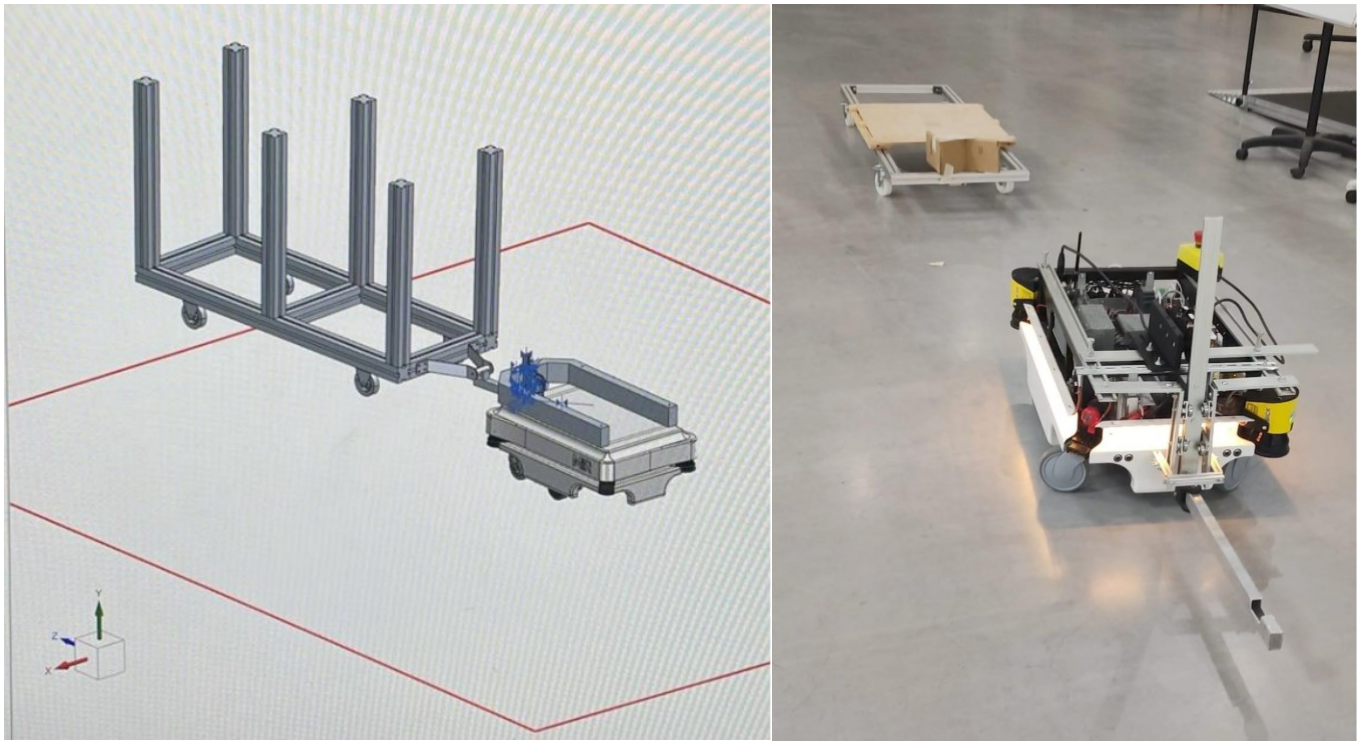
Keväällä 2025 valmistettiin MiR 100 -mobiilirobottiin kiinnitettävä kantomekanismi, joka soveltuu Finlandia-talon tuolipinojen kuljettamiseen. Valmistusprosessi alkoi mekanismin mallintamisella, minkä jälkeen osat leikattiin ja hitsattiin kasaan Metropoliassa. Tämän jälkeen mobiilirobotille kartoitettiin kulkureitti Finlandia talossa. Jotta robotti pystyy kuljettamaan tuolipinoja itsenäisesti kerrosten välillä, tarvittiin hissiin API-rajapinta, jonka avulla KONEen hissi pystyy keskustelemaan muiden ohjelmistojen kanssa. Näin mobiilirobotti pystyy tilaamaan hissin itselleen kulkiessaan kerrosten välillä.

Projekti toteutettiin monialaisena ja siihen osallistui myös teollisen muotoilun opiskelijoita Metropolian Arabia kampukselta. Muotoilijoiden tehtävänä oli työskennellä suunnittelijoiden kanssa ja luoda toteutettava esteettinen visio kokonaisuudelle. Päätavoitteena on piilottaa nosto-osan teollinen ulkonäkö ja korvata se ulkomuodolla, joka resonoi Finlandia talon arkkitehtuurin ja sisustuksen kanssa.

2. Projektin taustaa

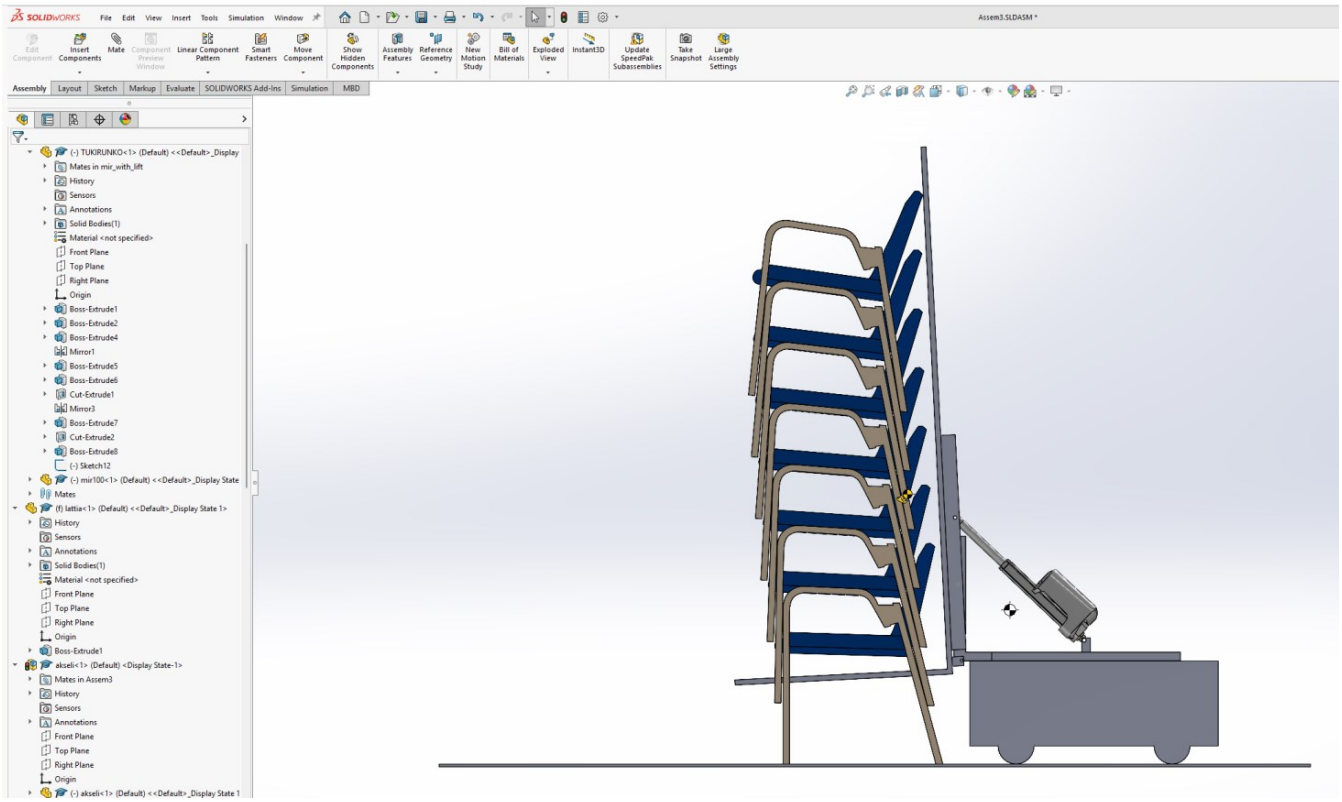
Finlandiatalo projektin alustaksi valikoitui MiR100 logistiikkarobotti, joka löytyi valmiiksi Metropolian Robo Garagelta. Syksyllä 2024 innovaatio projektiryhmä suunnitteli ja valmisti MiR100 perässä vedettävän kuljetus kärryn, joka kiinnittyi robottiin nousevan mekanismin avulla (Kuva 3). Peräkärry oli kuitenkin vaikea hallita ahtailla käytävillä ja projektin päätteeksi todettiin, että robotin päällä oleva nostomekanismi olisi helpommin hallittava. Tämän kokeilun jälkeen alkoi logistiikkarobottiin kiinnitettävän nosto-osan mekaniikkasuunnittelu opinnäytetyönä keväällä 2025.

Laitteesta visioitiin alkuun kaksi eri versiota. Toinen, jossa robotti vetäisi perässään kärryä, johon olisi lastattu jo valmiiksi tarvittavat huonekalut, Finlandia talon varastossa on näitä jo valmiiksi isommille pöydille ja niiden liikuttelu on ihmisille painon puolesta haaste. Jälkimmäinen lopulta toteutettu ratkaisu olisi nostaa jollain mekanismilla huonekalut robotin kyytiin, jolloin mahdolliset käyttökohteet suppenevat, mutta robotista tulee paljon ketterämpi ja helpommin ohjattava.



Kuva 3. Mallinnuskuva logistiikkarobotin perävaunusta sekä kuva nousevasta kiinnitysaisasta.

Kantolaitteen tehtävä on nostaa autonomisen robotin kyytiin tuolipinoja, joita se siirtää Finlandia-talon kiinteistössä hissiä käyttäen konferenssi- ja varastotilan välillä. Tässä suunnitteluraportissa selvitään nosto-osalle asetettuja vaatimuksia, sen painokeskipistelaskentaa ja kehitysprosessia. Toimivan konseptin saavuttamiseksi käytetään prototyyppien valmistuksen ja geometrisen suunnittelun yhdistelmää.



Kuva 4. Kantolaitteen mekaniikan suunnittelua SolidWorks-ohjelmassa.

2.1. MiR100-robotti

MiR100-robotin (Kuva 5) on kehittänyt tanskalainen Mobile Industrial Robots -yritys, jonka tuotteet keskittyvät automatisoituihin robotiikkaratkaisuihin. MiR100 on suunniteltu toimimaan autonomisesti logistisissa kuljetustehtävissä. Robotin kantokyky on max. 100 kg. Liikkumista ja turvallisuutta varten robotti on varustettu LiDAR- ja ultraäänisensoreilla, joilla se pystyy kartoittamaan ympäristöään ja välttämään törmäyksiä. Laitteen yläpinnan reunojen vieressä on neljä kiinnityskohtaa käyttäjän nosto- ja/tai kantomekanismin kiinnitystä varten. M10 kierteillä olevat kiinnityskohdat tarjoavat modulaarisen pohjan, johon käyttäjä voi kiinnittää tarvitsemansa lisäosan. Robotin päädyissä on kaksi rullapyörää ja keskellä kaksi rengasta sen liikkumista varten.

Kuva 5. MiR100-robotti (S&P Automation & Robotics)

MiR100 tekniset tiedot:

- Massa: 62.5 kg
- Kantokyky: 100 kg
- Toiminta-aika: 10 h tai 20 km
- Max. nopeus: 5.6 km/h
- Mitat (pituus, leveys, korkeus): 850 mm, 600 mm, 350 mm

2.2. Linak LA36-karamoottori

LA36 (Kuva 6) on Linakin valmistama raskaiden kuormien liikuttamiseen tarkoitettu lineaariaktuaattori. Laitteessa oleva sähköinen tasavirtamoottori pyörittää ruuvimekanismia, joka tekee iskuliikkeen. Projektissa karamoottoria käytetään nostomekanismin liikuttamiseen. Suunnittelussa käytetty CAD-malli on Linakin internet-sivulta. (Linak, LA36.)

Kuva 6. Linak LA36-karamoottori (Linak, LA36)

Linak LA36 tekniset tiedot:

- Type: 362002000A01BA-611F30400NCS0006
- Item No: J66352
- Massa: 5 kg
- Max. työntö- ja vetovoima: 1700 N
- IP66
- Moottorin tyyppi: 24 V DC
- Iskunpituus: 200 mm

2.3. Finlandiatalon tuolit

Finlandia-talo käyttää konferenssisalissaan Alvar Aallon suunnittelemia puukangastuoleja (Kuva 8). Ne ovat valmistettu ja suunniteltu eksklusiivisesti Finlandia-taloa varten eikä niitä löydy muualta. Tuoleja on olemassa vain 1500 kappaletta. Harvinaisuuden ja Alvar Aallon muotoilun arvostuksen ansiosta tuolien arvo on äärimmäisen korkea. Koska kyseistä mallia ei enää valmisteta, Finlandia-talon henkilökunta tekee kunnossapitotyötä käyttääkseen tuoleja edelleen ylläpitäen Finlandia-talon esteettisen vision.

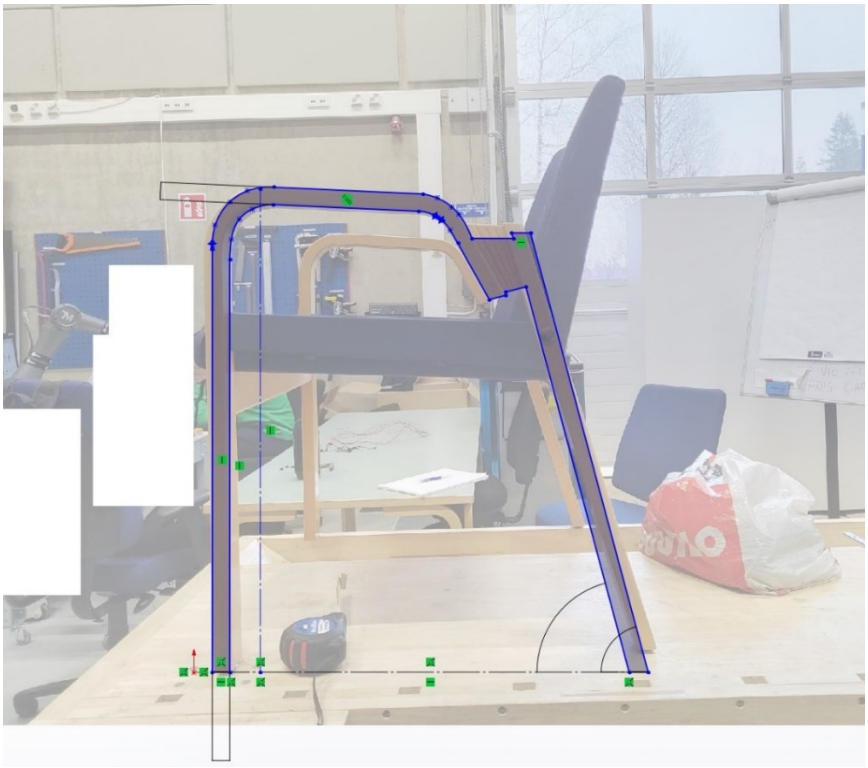
Kuva 7. Kongressisali (Finlandia-talo)

Korvaamattomuuden ja suuren arvonsa vuoksi tuolien vaurioittamisen välttämisestä tehdään projektissa painopiste. Tuolin puuosa on tehty taivutetusta vanerista ja pehmusteiden pinta kankaasta. CAD-malli tuolista Kuva 8) luotiin käyttäen fyysisiä mittoja ja SolidWorksin ”sketch picture” -työkalua (Kuva 9).

Tuolin tekniset tiedot:

- Massa: 5 kg
- Mitat (pituus, leveys, korkeus): 555 mm, 573 mm, 760 mm

Kuva 8. Finlandiatalon kongressisalintuoli sekä valmis CAD-malli tuolista.



Kuva 9. Tuolin CAD-mallin luominen SolidWorks-ohjelmalla.

3. Tekninen toteutus

3.1. Robotin tuolinnosto-osan mekaniikkasuunnittelu

Tässä osiossa käsitellään ja avataan MiR100-robotin päälle kiinnitettävän nostomekanismin suunnitteluprosessia.

Tarkkaa painokeskipistelaskentaa varten MiR100-robotista on tehty yksinkertaistettu CAD-malli. Alkuperäisessä mallissa on ulkomuoto, mutta ei sisällä materiaaliominaisuuksia. Malli on siis ontto. Tämän takia painokeskipistelaskennassa saadaan harhaanjohtava tulos, sillä kaikkia osia ei oteta laskentaan mukaan. Yksinkertaistetulla mallilla saadaan realistisempi kuvaus MiR100-robotin painokeskipisteestä. Robotin edessä ja takana on rullapyöriä, joiden asento ja painonjakauman kohta lattian kanssa muuttuu. Turvallisuuden vuoksi yksinkertaistetussa mallissa rullapyörät ovat asennossa, jossa ne osoittavat sisäänpäin. Tässä asennossa se on kaikista altistunein kaatumiselle, sillä tukipisteiden välinen pinta-ala on kaikista pienin.

Pienemmällä robotilla olisi mahdollista ajaa tuolikasan alle ja nostaa se pystysuoralla mekaanisella liikkeellä kantoon. Leveytensä takia, tämä ei ole MiR100-robotilla mahdollista. Suunnittelun tehtävänä on siis saada 200 mm karamoottorin iskunpituudella tuolikasa robotin edestä sen päälle. Tämän saavuttamiseksi käytetään vipuvaikutusta.

3.2. Nosto-osan vaatimukset

Nosto-osalle on asetettu seuraavat vaatimukset, jotta kuljetusrobotti ja nostomekanismi voisivat toimia Finlandia-talossa:

- Tuolikasan alimman kohdan ja lattian väli kannon aikana tulee olla vähintään 200 mm. Tämän tarkoituksena on antaa tarpeeksi tilaa MiR100-robotin LiDAR-sensoreille, mikä on välttämätöntä sen toiminnalle.
- Tuolikasan korkeus maasta tulee olla alle kaksi metriä kannon aikana. Finlandia-talon kiinteistön ovet sekä hissin sisäänkäynti rajoittavat kokonaisuuden maksimikorkeutta.
- Nosto-osan ja tuolipinon yhteinen massa tulee olla MiR100-robotin kantokyvyn (max.100 kg) sisällä.
- Robotti ei ole vaarassa kaatua nosto-osaa kantaessa tai käyttäessä.

3.3. Painokeskipisteen huomioiminen suunnittelussa

Suunnittelun keskeisenä osana ja tavoitteena on lopputuloksen optimaalinen painopiste. Tämän tarkoituksena on estää kaatuminen välttäen materiaalivaurioita ja ihmisten loukkaantumista. MiR100-robotin kuusi tukipistettä ovat sen neljä rullapyörää ja kaksi rengasta.

Optimaalinen painokeskipiste on ominaisuuksiltaan sellainen, joka on tukipisteiden välisen pinta-alan yläpuolella, sillä ”kappale kaatuu, kun sen painopisteen kautta kulkeva luotisuora joutuu alustan

tukipisteiden rajaaman alueen ulkopuolelle” (Inkinen & Tuohi 2006: 274). Kokonaisuuden painopisteen sijaintiin tulevat vaikuttamaan rakenne- ja materiaalivalinnat, joiden avulla tavoitellaan haluttua tulosta.

3.4. Prototyyppini

Nosto-osan suunnittelu ja kehittäminen on tehty prototypoinnilla. Siinä halutusta tuotteesta tehdään alkeellisia versioita, joita käytetään selvittämään epäkohtia ja ratkaisemaan niitä. (Protoype; What are Prototypes?). Prototyypit esitetään raportissa kronologisessa järjestyksessä.

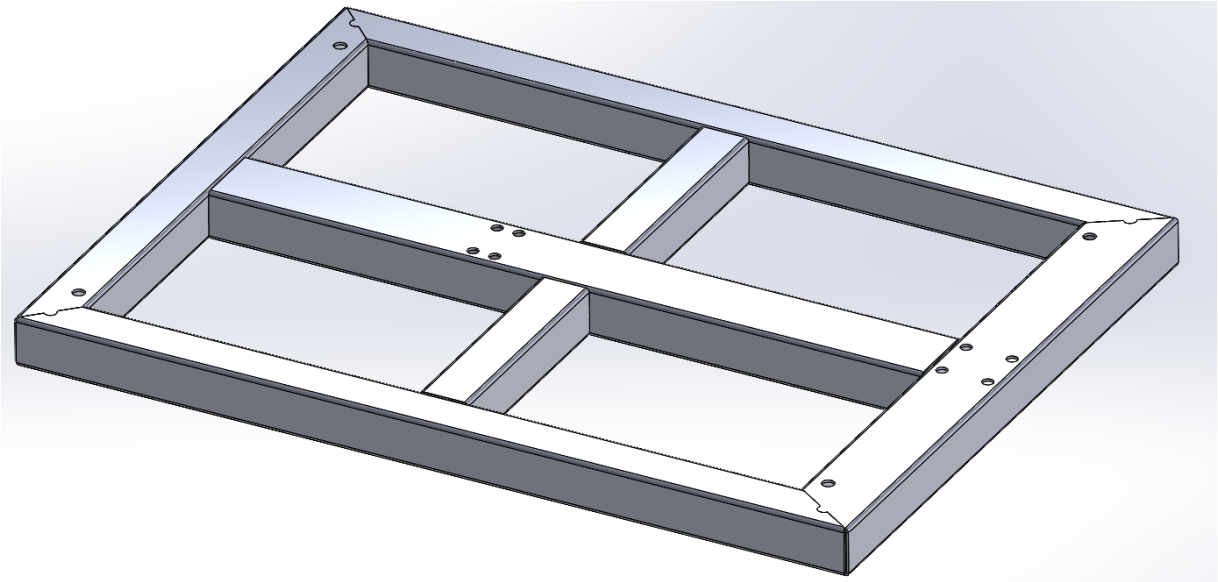
Suunnittelussa haluttiin ensimmäiseksi luoda karkea prototyyppi, jossa mekaaniset tavoitteet on saavutettu. Tämän jälkeen suunnittelussa siirryttäisiin rakenteen yksityiskohtiin. Näitä ovat materiaalivalinnat ja kiinnitysratkaisut.

Dassault Systèmesin valmistamaa SolidWorks-tietokoneavusteista suunnittelu- ja simulointiohjelmaa käytetään luomaan prototyyppimalleja, joista kehitetään valmis lopputulos. SolidWorks-ohjelmisto tarjoaa erinomaiset työkalut mekaaniselle suunnittelulle. Tärkeimmät työkalut, joita projektissa käytetään ovat seuraavat:

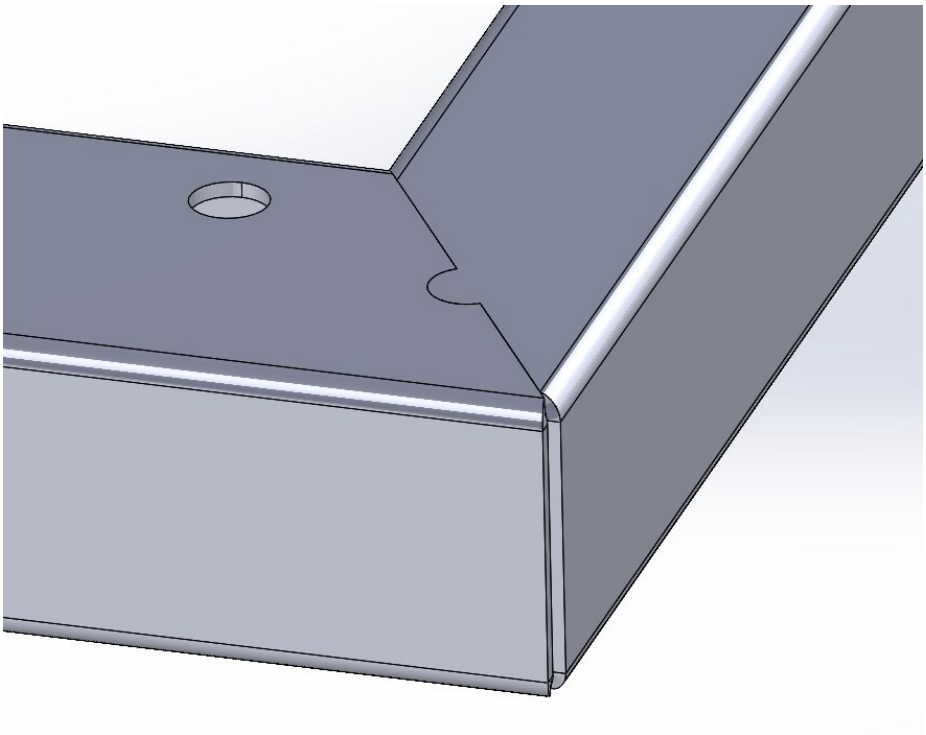
- Painokeskipisteen laskeminen. SolidWorks pystyy laskemaan ja näyttämään haluttujen osien painokeskipisteet kokoonpanossa. Projektissa painokeskipisteen tietäminen on oleellista, koska sitä käytetään selvittämään, ovatko nostolaite ja robotti vaarassa kaatua.
- SolidWorksin sisäisiä kokoonpanoja voi Liitos(mate) työkalulla yhdistää halutulla tavalla. Mate voi luoda niveliä, kosketuspintoja, kohtisuoruuksia, määrättyjä kulmia ja etäisyyksiä.
- Liikkeen ja mekaniikan simulointi. Kun alikomponentit ovat valmiit ja kiinnitetty mate-työkalulla, niitä voidaan liikuttaa. Liikuttamisen avulla pystyy näkemään nostoliikkeen, tarkistamaan sen toiminnan ja mittatyökalulla (measure) laskemaan täyttyvätkö otsikossa 3.1 asetetut mittavaatimukset.
- Osien muuttaminen kokoonpanossa. Vaikka alikomponentit olisivat kiinnitetty toisiinsa, niitä voidaan silti muokata siten, että muodonmuutoksista riippumatta ne siirtyvät ja adaptoituvat kokoonpanoon muutosten jälkeen. Tämä on erityisen tärkeää, sillä nosto-osan haluttu mekaaninen liike saavutetaan muuttamalla komponenttien muotoja ja mittoja.
- Elementtimenetelmä (finite element analysis eli FEA): FEA on suunnitteluanalyysissa käytettävä työkalu, jonka avulla voidaan määrittää kappaleiden, kokoonpanojen ja rakenteiden ominaisuuksia (Kurowski 2004: 1). Näitä voivat olla mm. sähkömagnetismi, lämpölaajeneminen, lujuus ja rasituksenkesto. Suunnitteluvaiheessa pystytään tarkistamaan täyttyvätkö tiettyjen ominaisuuksien vaatimukset ennen kokoonpanon toteuttamista. Kyseisessä tapauksessa FEA-työkalulla lasketaan nosto-osan rasituksenkestoa sen kantaman kuorman alla. Tavoite on tarkistaa, että nosto-osan komponentit ovat tarpeeksi vahvoja kestämään niihin kohdistuvat painokuormat. (Simscale 2024.) Tätä aihetta jatketaan riskianalyysiotsikossa 5.3.

3.5. Kiinnitysrunko

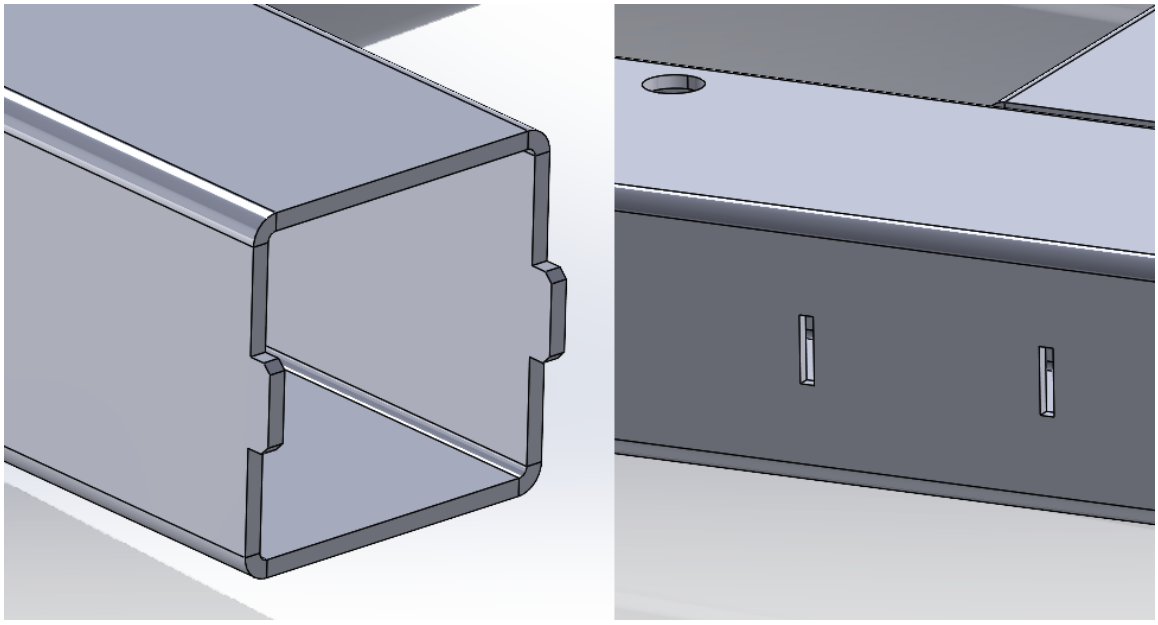
Runko (Kuva 10), joka kiinnitetään MiR100-robottiin, on perusta nostolaitteelle. Sen tulee ylettää neljään kiinnityskohtaan sekä kestää siihen välittyvät veto- ja nostovoimat. Rungossa käytetään 40 x 40 mm teräsputkea, paitsi pääty- ja keskiosissa. Näissä kahdessa putkien mitat ovat 40 x 60 mm, jotta niihin mahtuisi nivelten leveät kiinnitykset. Teräsputkien paksuus on 2 mm. Rungon reunakohtiin on leikattu 10 mm reiät M10 muttereiden kiinnitystä varten. Keskilinjaa pitkin on 9 mm reikiä M6 niittimuttereiden kiinnitystä varten. Teräsputkien rakenteeseen on luotu kohdistusmuotoja, jotka auttavat kokoonpanossa ja toleranssien saavuttamisessa (Kuva 11 ja Kuva 12).



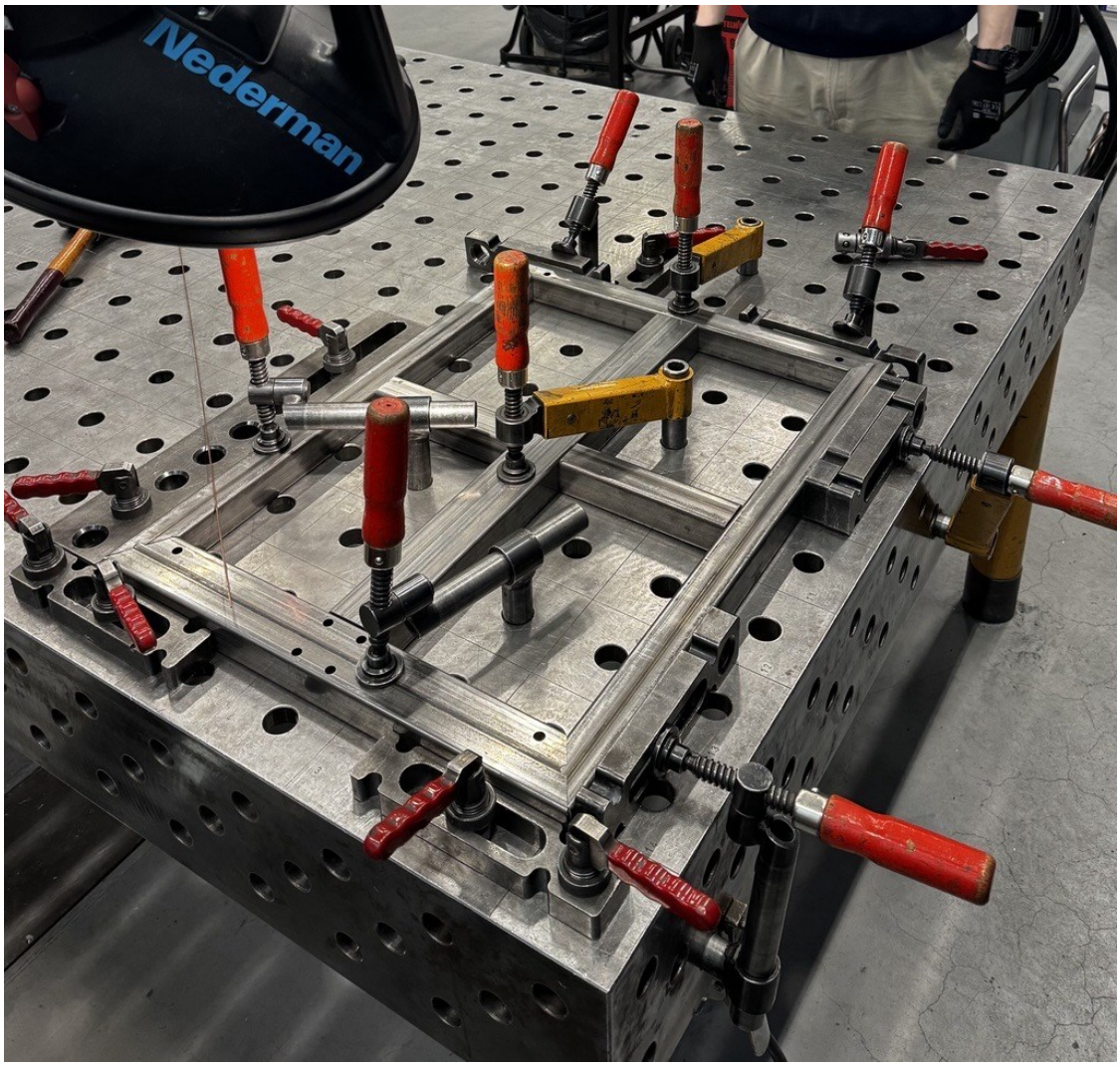
Kuva 10. Nostolaitteen kiinnitysrunko.



Kuva 11. Kiinnitysrungon jiirikulmien "palapeli"-kiinnitys.



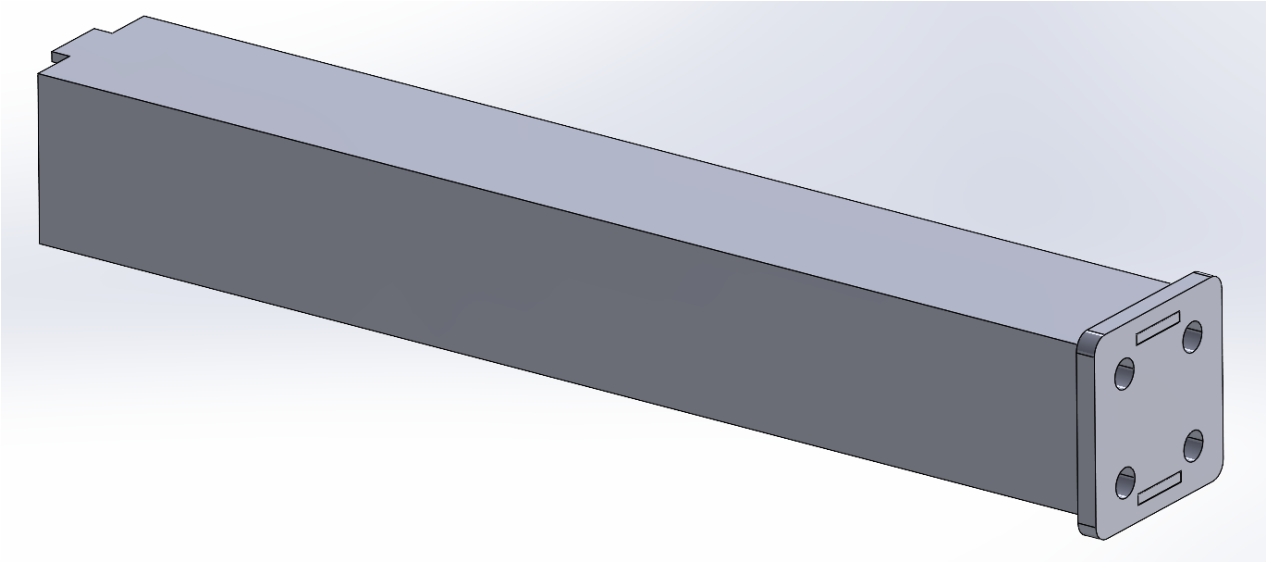
Kuva 12. Kehikoiden kohdistusnastat.



Kuva 13. Kantomekanismin rungon hitsaus JIG-pöydällä.

3.6. Kiinnitysvarsi

Kiinnitysvarsi (Kuva 14) yhdistää L-kappaleen alaosan ja kiinnitysrungon. Se on tehty 50 x 50 x 350 mm teräsputkesta, jonka vahvuus on 3 mm. Kiinnitysvarren nivelpäätyyn hitsataan kiinni 60 x 60 x 6 mm teräslevy. Feston SUA-32 nivelosaa varten levyssä on neljä 9 mm reikää M6 niittimuttereita varten. Levyn reunoja on loivennettu 5 mm säteellä. Kiinnitysvarren päädyissä on myös kohdistusnastat toleranssien ja hitsausprosessin helpottamisen vuoksi.



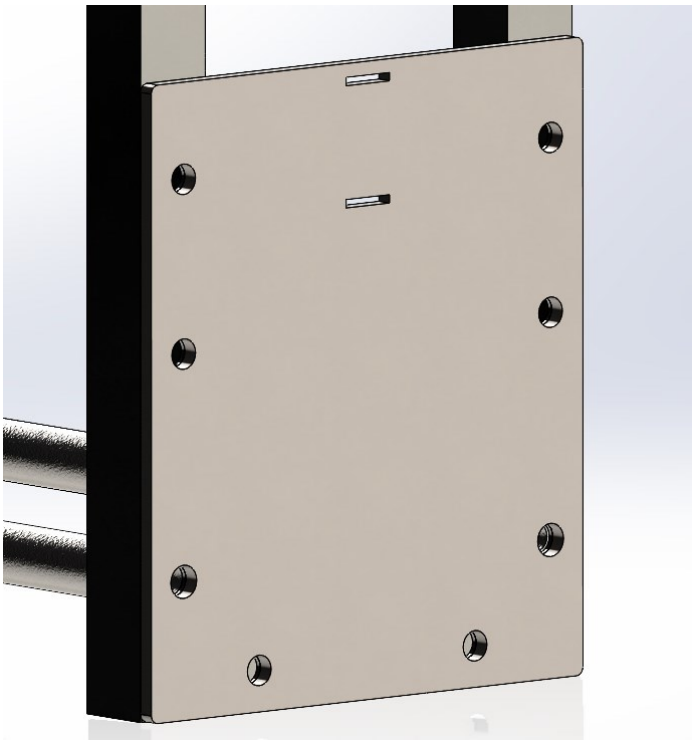
Kuva 14. Kiinnitysvarsi.

3.7. L-osa ja selkälevyt

L-osa tulee nostamaan tuolit maasta ja kantamaan ne siirtymisen aikana. Osan taakse asennetaan selkälevyt, joihin yhdistyvät kiinnitysvarsi ja karamoottorin jatkovarsi. Runko koostuu pysty- (30 x 30 x 1300 mm) ja vaakasuorista (30 x 30 x 200 mm) teräsputkista, joiden paksuus on 2 mm.

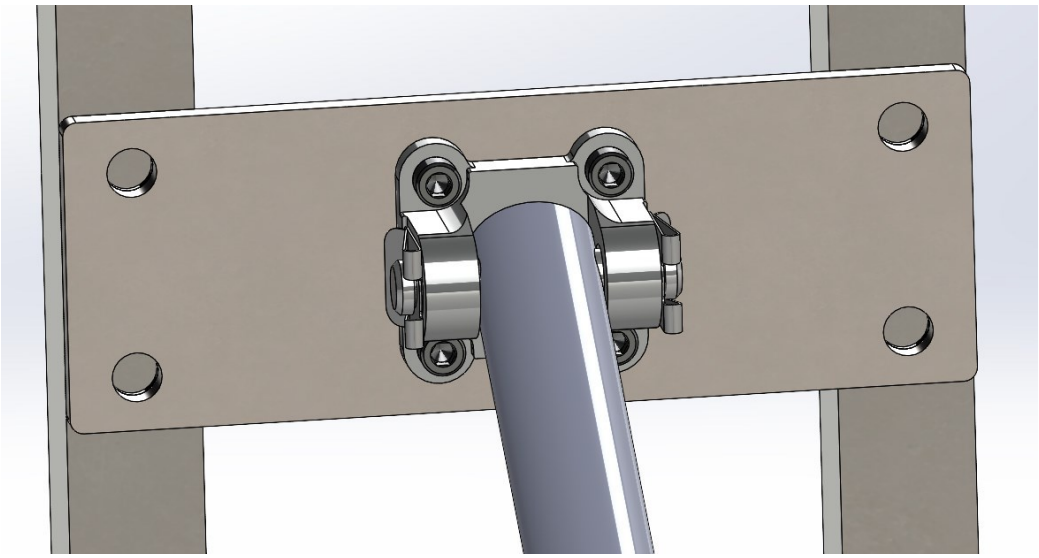
Alempi selkälevy (Kuva 19) on mitoiltaan 230 x 200 x 6 mm. Se sisältää yhteensä kahdeksan reikää, joiden kautta se kiinnitetään L-osaan:

- Kuusi 8.2 mm reikää, joiden läpi M8 ruuvit kiinnittyvät rungon niittimuttereihin.
- Kaksi 12.2 mm reikää, joiden läpi M12 ruuvit kiinnittyvät putken sisällä oleviin kierteisiin.



Kuva 15. Alempi kiinnityslevy.

Ylempi kiinnityslevy (Kuva 16) on mitoiltaan 200 x 70 x 6 mm. Se sisältää neljä 8.2 mm reikää L-osaan kiinnitystä varten, ja neljä 9 mm reikää niveltä varten.



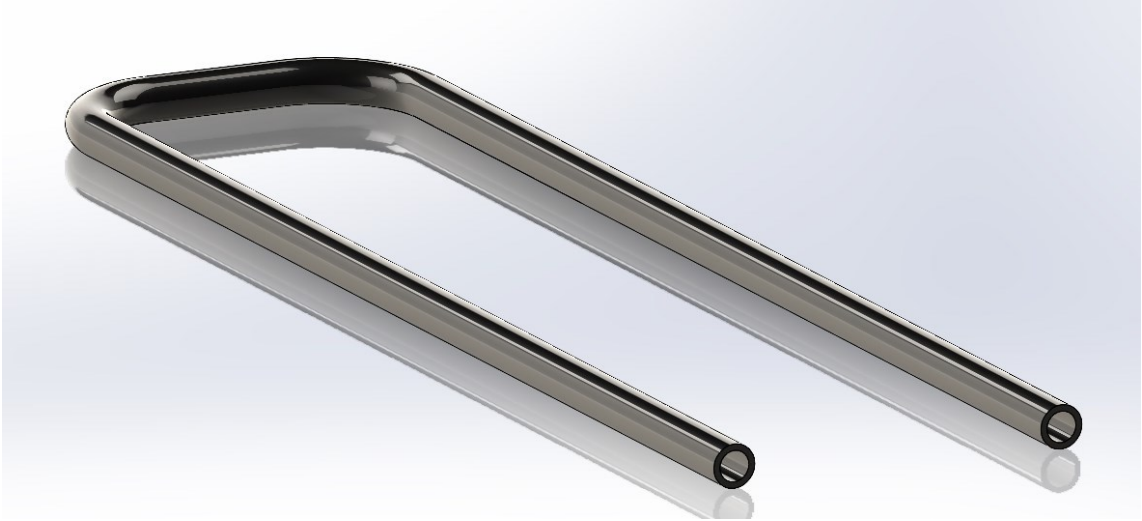
Kuva 16. Ylempi kiinnityslevy SNCB-40 nivelen kanssa.

3.8. Putkiosa

Tuoleihin luodaan kontakti alhaalta putkiosalla (Kuva 17):

- Putken pituus suorana on 1685 mm ja taivutuskohtien säde on 50 mm.
- Taivuttamisen jälkeen muodon ulkoleveys on 191,3 mm ja pituus 710,65 mm
- Ulkohalkaisija on 21,3 mm ja vahvuus 3,2 mm

Putken päätyihin asennetaan M12 kierreholkit, jotta se voidaan ruuveilla kiinnittää L-osan runkoon ja alempaan selkälevyyn.

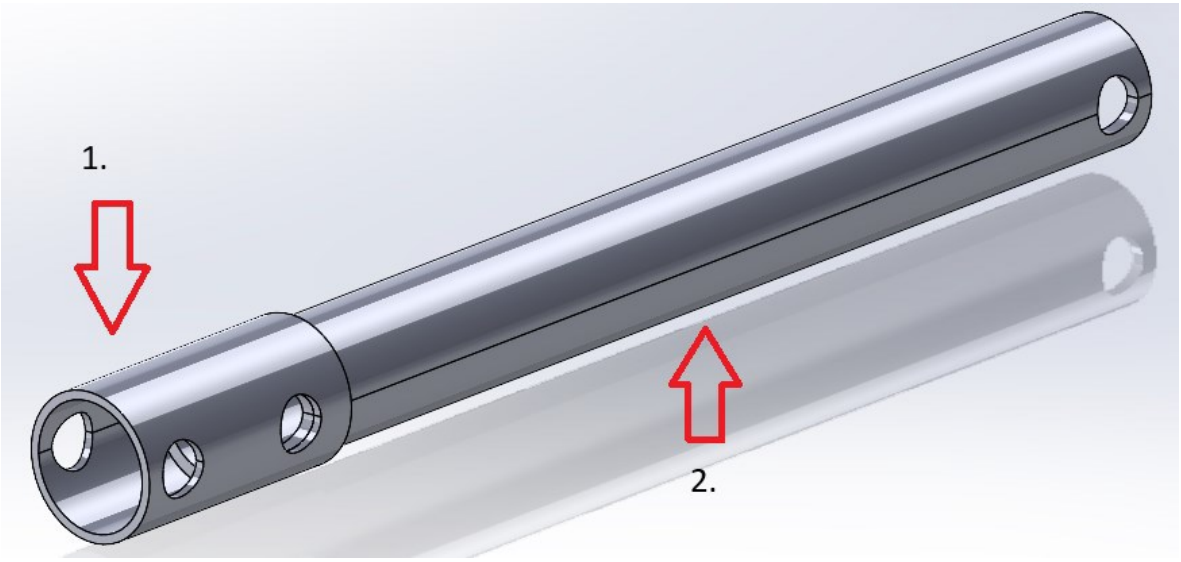


Kuva 17. Putkiosa

3.9. Karamoottorin jatkovarsi

Jatkovarsi (Kuva 18) koostuu kahdesta osasta:

- Karamoottorin kiinnitysosa: 60 x 32 mm putki, 2 mm vahvuudella
- Varsi: 265 x 28 mm putki 3 mm vahvuudella



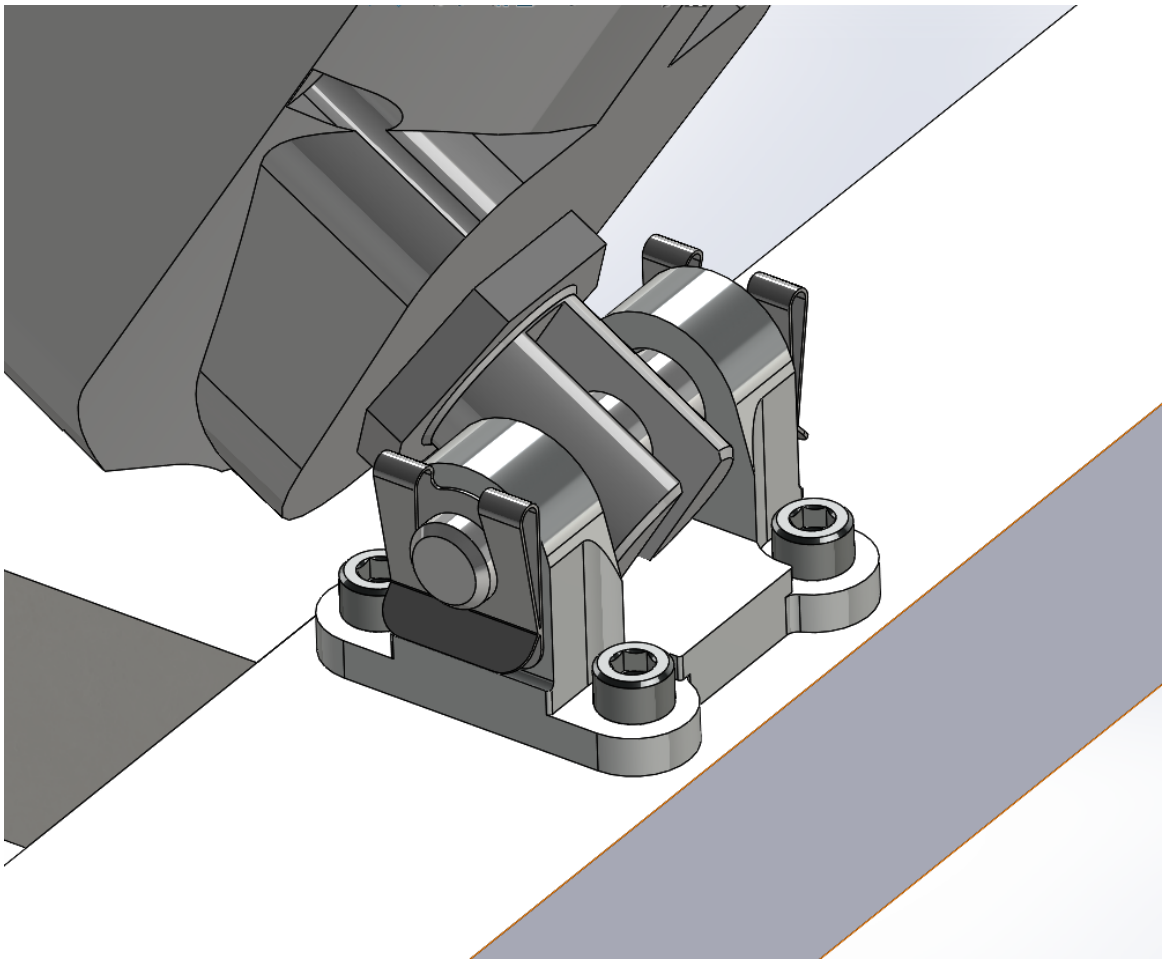
Kuva 18. Karamoottorin jatkovarsi.

Varren ulkopintaa koneistetaan kiinnitysosan liitosalueella 0,5 mm, jotta osat mahtuvat liittymään. Jatkovarren suunnittelussa luo vaikeuksia kiinnitettävien kohtien erikokoiset mitat ja materiaalivalikoiman vaje. L-osan nivelen välinen kiinnitystila on leveydeltään 28 mm ja karamoottorin kiinnitysosan ulkohalkaisija myös 28 mm. Tämän takia jatkovarren päätyjen halkaisijat eivät voi olla saman-kokoiset. Kiinnitysosan hankintapäätöksessä jouduttiin tyytymään 32 mm ulkohalkaisijan putkeen, vaikka

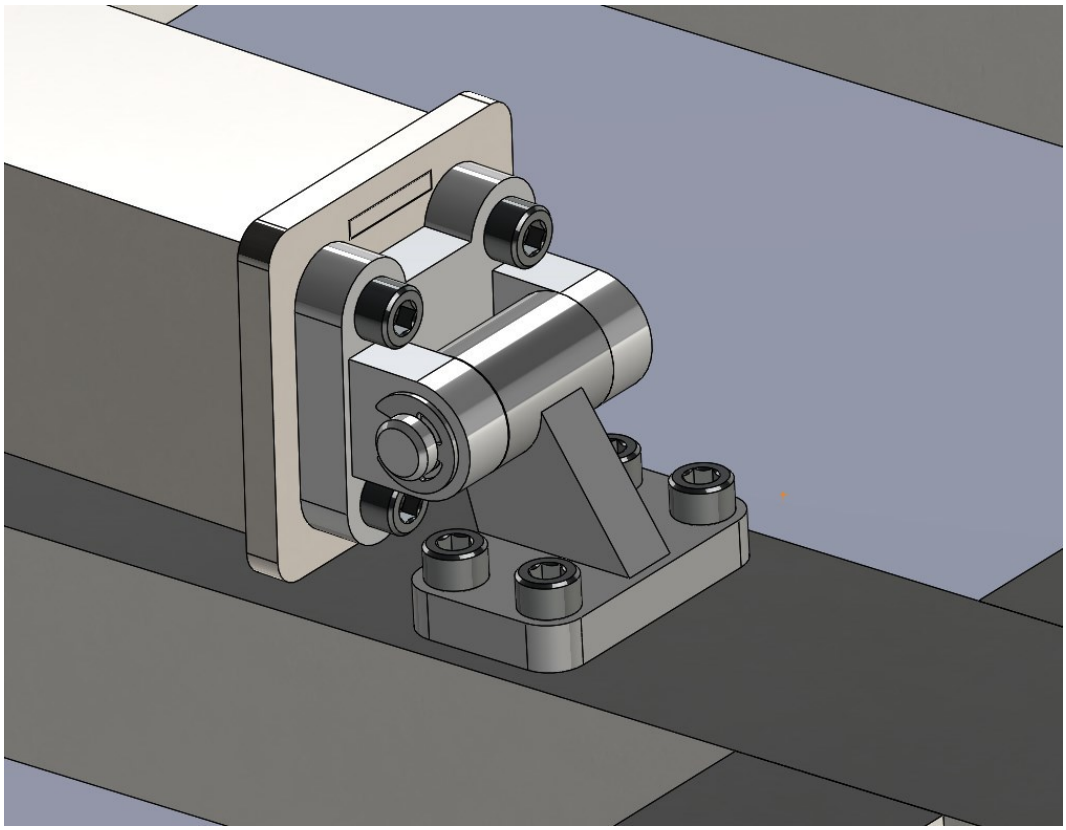
optimaalinen mitta olisi ollut 34 mm. Tämä johtui halutun putken pitkästä toimitusajasta, mikä ei olisi toiminut projektiaikataulun kanssa. Työryhmä tulee improvisoimaan kiinnitysosalle ylimääräisen ulkorakenteen vahvuutta varten. Osissa on 12 mm reikiä lukkotappiliitoksia varten, joilla ne kiinnitetään toisiinsa ja muihin komponentteihin.

3.10. Nivelosien komponentit

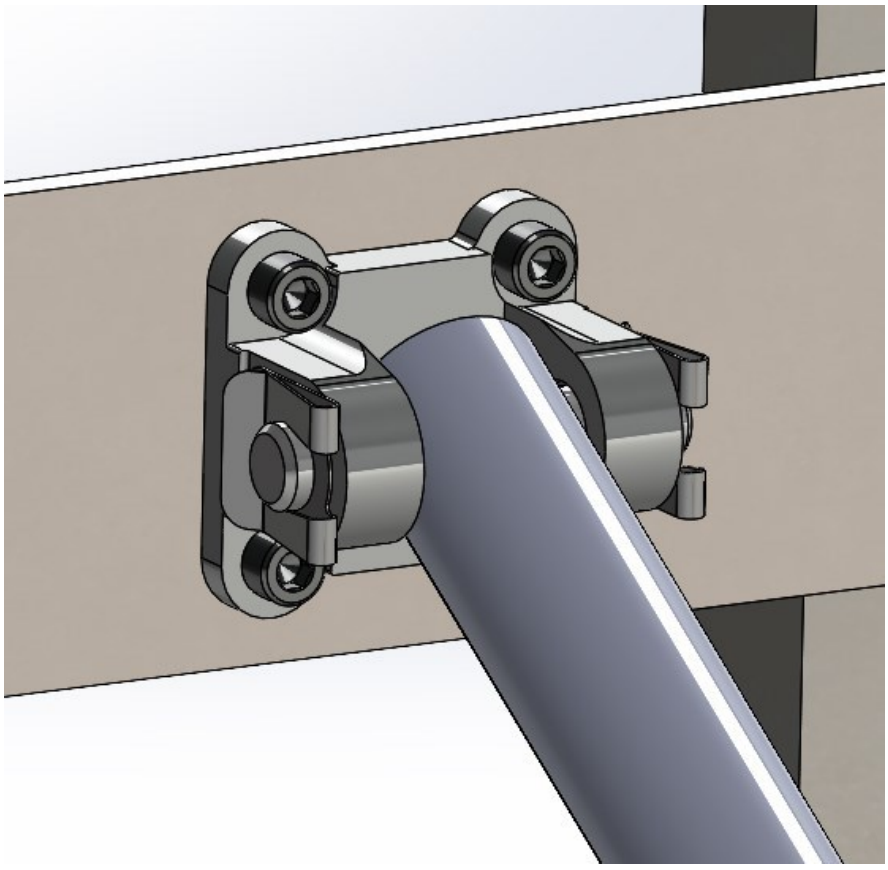
Valmiissa prototyypissä on kolme nivelkohtaa, joista kaksi on karamoottorin päädyissä. Nivelkomponenteiksi valittiin Feston valmistamat ratkaisut. Feston nivelet tarjoavat helpon asentamisen metalliputkiin ilman hitsauksen tarvetta ja ovat suunniteltu kestämään raskaita työntö- ja vetovoimia. Teräsputkiin pyydettiin valmistajalta 9 mm halkaisijan reiät, joihin asennetaan M6 X 16 MM niittimut-terit. Nivelet kiinnitetään niittimuttereihin M6 ruuveilla. Karamoottorin ja rungon välistä SNCB-40 nivelkomponenttia (Kuva 19) tullaan koneistamaan, jotta se sallii halutun liikkuvuuden. Festo tarjoaa asiakkailleen CAD-malleja komponenteista, joita käytettiin suunnittelussa ja lopullisessa mallissa (Festo, CAD-mallit).



Kuva 19. Karamoottorin kiinnitys runkoon: SNCB-40 (koneistettu)



Kuva 20. Kiinnitysvarren ja rungon nivel: SUA-32 & CRLNG-32



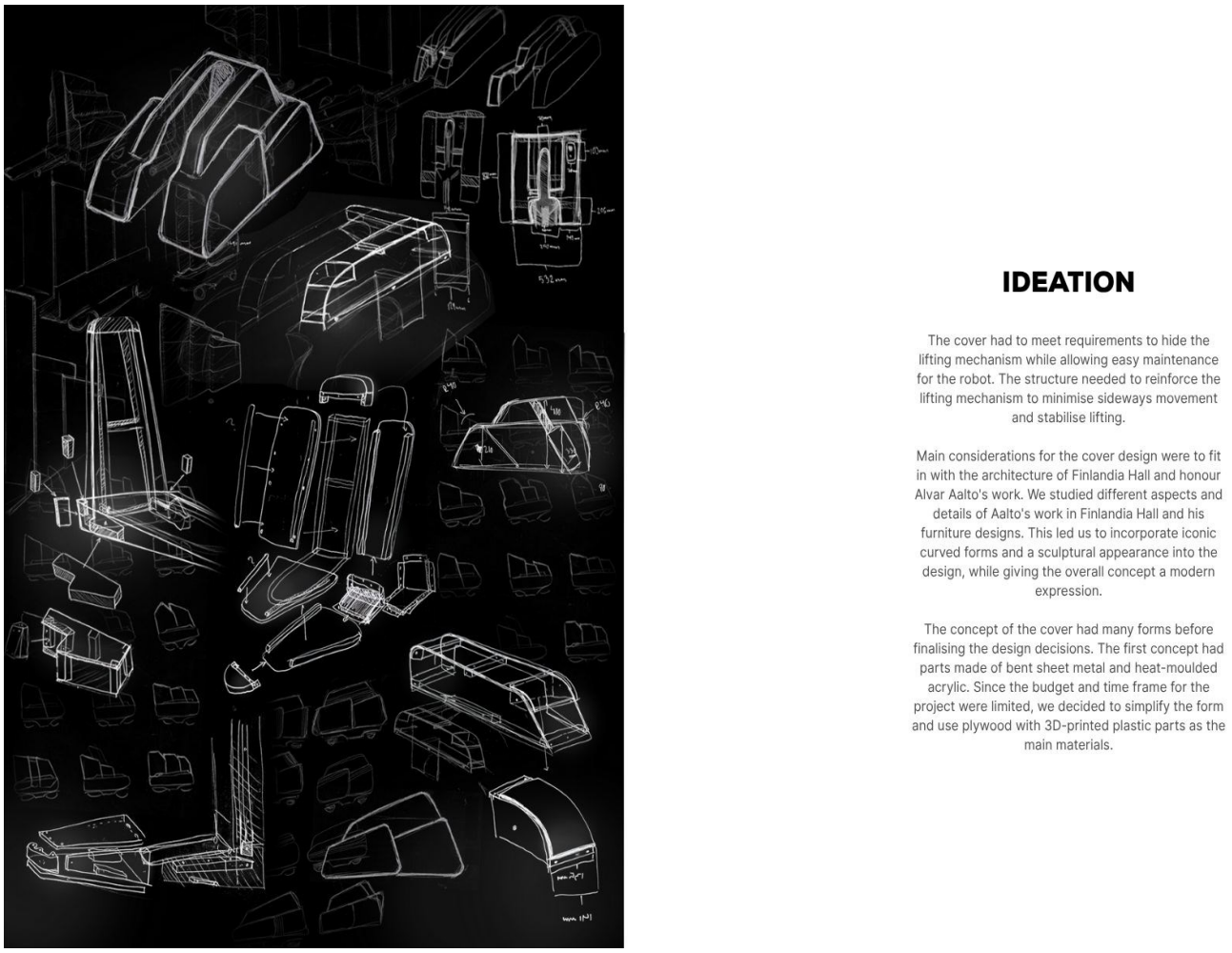
Kuva 21. Jatkovarren kiinnitys selkälevyyn: SNCB-40

4. Muotoilu

Kuoren suunnittelussa vaatimuksena oli, että se peittää nostomekanismin ja mahdollistaa helpon huollettavuuden robotille. Rakenteen täytyi myös tukea nostomekanismia sivuttaisliikkeen minimoimiseksi ja nostamisen vakauttamiseksi.

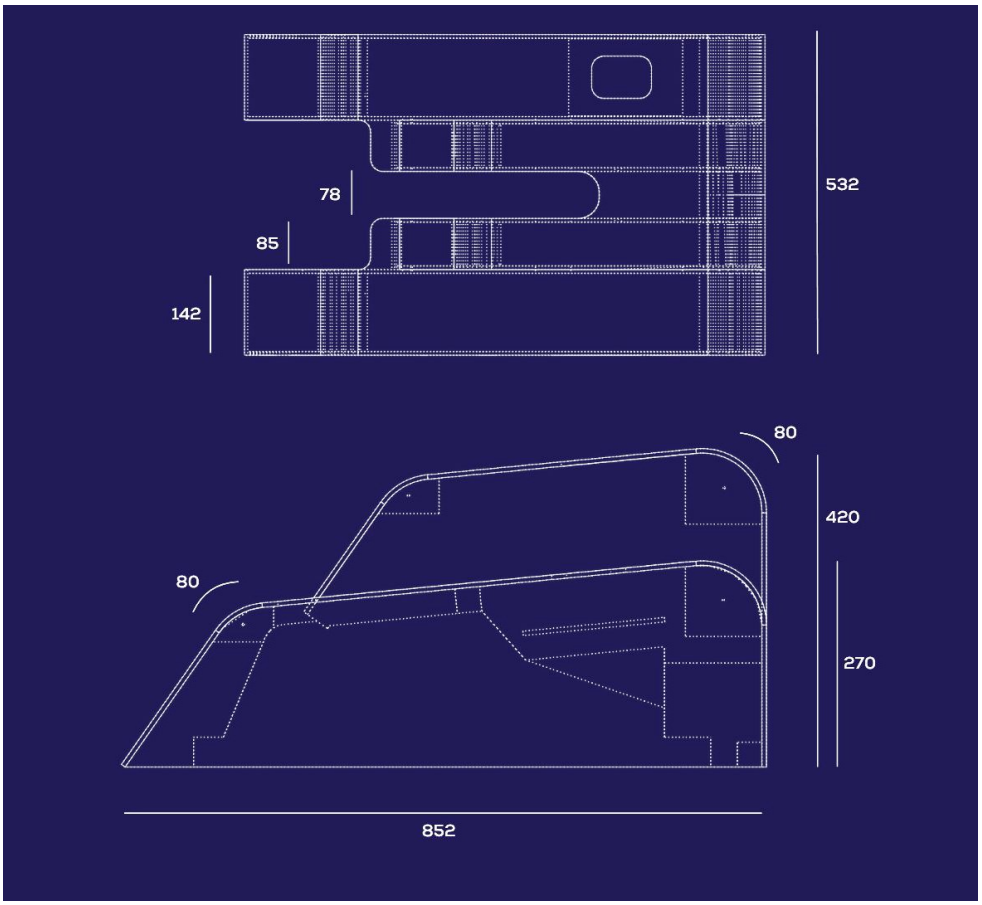
Kuorirakenteen suunnittelussa keskeistä oli sovittaa se Finlandia-talon arkkitehtuuriin ja kunnioittaa Alvar Aallon kädenjälkeä. Tutkimme Aallon töitä, Finlandia-talon yksityiskohtia ja hänen suunnittelemien huonekalujen muotoilua. Tämä inspiroi meitä tuomaan suunnitteluun ikonisia kaarevia muotoja ja veistoksellista ilmettä, samalla kun kokonaisuuteen haettiin modernia ilmaisua.

Kuoren konsepti kävi läpi useita muotoja ennen lopullisia valintoja. Ensimmäinen versio sisälsi taivutetusta pellistä ja lämpömuovatusta akryylista valmistettuja osia. Projektin rajatun aikataulun vuoksi päätimme yksinkertaistaa muotoa ja käyttää päämateriaaleina vaneria ja 3D-tulostettuja muoviosia.



Kuva 22. Konseptivaiheen luonnoksia.

Kuva 23. Muotoilijaopiskelija Lauri Ojanen työskentelemässä CAD-ohjelman parissa



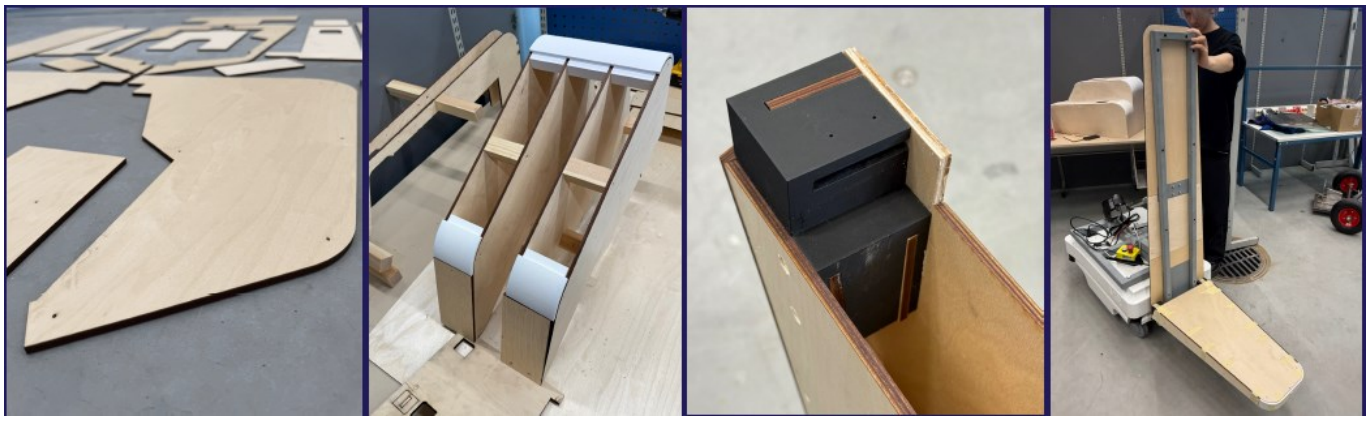
Kuva 24. Kuoren teknisiä mittoja

4.1. Rakennusvaiheet

3D-tulostuksen avulla saatiin lisättyä rakenteellista vahvuutta ja samalla muotoiltua pyöreät ulkopinnat, mikä teki kokoonpanosta helpomman ja nopeamman.

Kuva 25. 3D-printattuja osia.

Vanerikomponentit koostuvat monimuotoisista muodoista, jotka on suunniteltu asettumaan tarkasti paikoilleen kokoonpanossa. Tämän ansiosta ulkokuori voitiin kiinnittää suoraan runkoon ilman ruuveja, mikä helpottaa huoltoa.

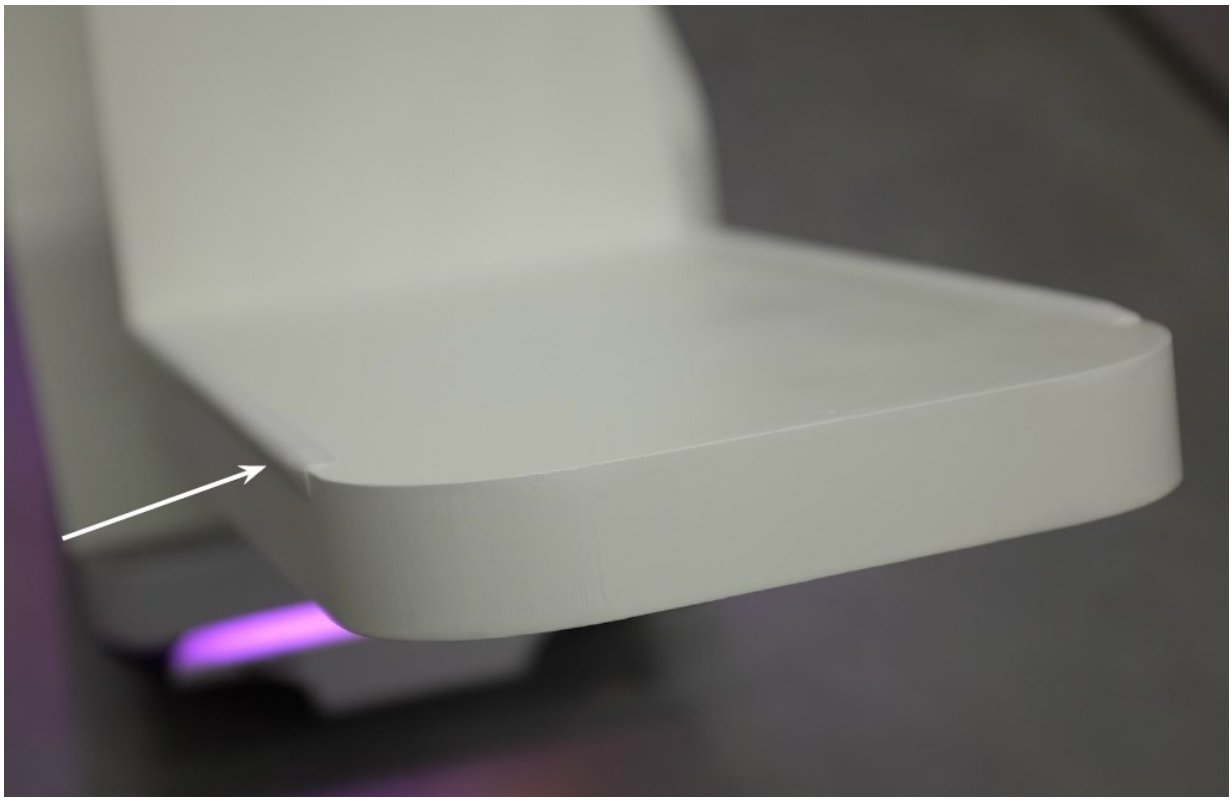


Kuva 26. Kuvia kokoamisvaiheesta.

Kuva 27. Kaikki kuoren osat kitattiin ja hiottiin huolellisesti ennen maalaamista.

Kuva 28. Kuoren osien kokeilua MiR100 robotin päälle ennen maalausta.

Kuva 29. Osat ruiskumaalattiin ja lakattiin Metropolian pintakäsittely laboratoriossa.



Kuva 30. 3D-tulostetut suojakaiteet suojaavat tuoleja ja nostimen pintaa.

Kuva 31. Valoelementti asennettuna.

Hienovarainen valoelementti lisättiin tuomaan robotille läsnäolon tuntua. Viimeistelynä levitettiin kaksi kerrosta pohjamaalia, lopullinen sävytetty maalipinta sekä suojaava lakkakerros.

Kuva 32. Valmis kuori kasattuna MiR100 robotin päälle.

5. MiR100-mobiilirobotin integrointi KONEen hissijärjestelmiin automatisoitua logistiikkaa varten

Tämä osio esittelee automaation suunnittelun ja toteutuksen autonomiseen logistiikkaan Finlandia-talossa, hyödyntäen MiR100-mobiilirobotin ja KONEen hissijärjestelmien integraatiota. Yhdistämällä molempien alustojen tarjoamat REST- ja WebSocket-rajapinnat kehitettiin middleware-sovellus, joka mahdollistaa MiR100-robotin itsenäisen esineiden kuljetuksen useiden kerrosten välillä ilman ihmisen väliintuloa. Järjestelmä hakee reaaliaikaiset sijaintitiedot ja tehtävän tilat MiR-robotista, synkronoi toiminnot hissin saatavuuteen ja oven tiloihin KONEen digitaalisilta rajapinnoilta sekä suorittaa rakennuksen sisäiset toimitustehtävät ennalta määritellyillä reiteillä.

Kattavat testaukset suoritettiin KONEen virtuaalisissa hissiympäristöissä ennen sertifiointia fyysiselle käyttöönotolle. Tulokset osoittavat, kuinka saumaton rajapintapohjainen integraatio autonomisten robottien ja edistyneiden hissijärjestelmien välillä voi optimoida kiinteistölogistiikkaa, parantaa toiminnan tehokkuutta ja tukea tulevaisuuden älyrakennussovellusten kehittämistä, Finlandia-talon toimiessa käytännön referenssinä.

Vaikka yksikerroksisia autonomisia mobiilirobottijärjestelmiä (AMR) toteutetaan yleisesti, usean kerroksen välinen kuljetus on edelleen haastavaa ja vaatii tarkan ja synkronoidun toiminnan robottien ja hissien välillä. Hissien manuaalinen käyttö rajoittaa robottipohjaisen logistiikan tehokkuutta ja skaalautuvuutta.

Tämän integraatioprosessin päätavoitteet ovat:

- Integroi MiR100-mobiilirobotti KONE-hisseihin niiden API-rajapintojen avulla, mahdollistaen autonominen monikerrosmateriaalin siirto Finlandia-talossa.
- Kehitä sovellus, joka lukee reaaliaikaisesti sijainti- ja tehtävätiedot MiR-robotilta, hallitsee hissikutsut oikea-aikaisesti ja synkronoi robotin toiminnot hissin oven tilojen ja kerrokselle saapumisen kanssa.
- Varmista integroidun järjestelmän toimivuus sekä virtuaalisissa että fyysisissä hissiympäristöissä, varmistaen luotettava viestintä, turvallisuus ja sertifiointi käytännön operatiivista käyttöä varten.

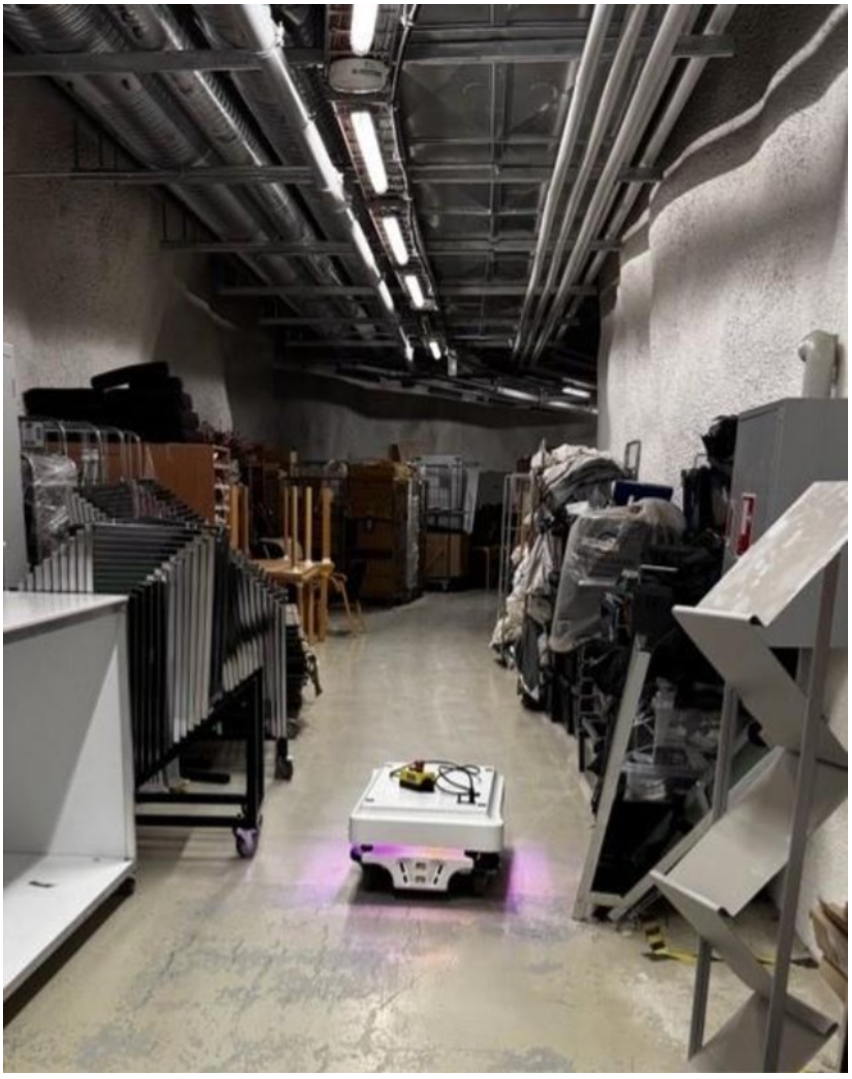
6. MiR 100-robotin API:n käyttö

MiR100-robotti toimii tässä integraatiossa logistiikan alustana hyödyntämällä omaa REST-rajapintaansa, joka mahdollistaa yksityiskohtaisen kontrollin tehtävien, navigaation ja tilanvalvonnan osalta. Rajapintojen hyödyntäminen mahdollistaa vankan automaation, reaaliaikaisen tilaseurannan sekä saumattoman yhteistoiminnan hissien kanssa.

Osiossa käydään läpi menetelmät ja logiikka, joilla viestitään MiR-robotin kanssa, haetaan olennaiset tiedot sekä suoritetaan tehtävät synkronoidusti rakennuksen muiden toimintojen kanssa. MiR:n tehtävät ja kartat tulee laatia robotin oman käyttöliittymän kautta.

6.1. MiR 100 -käyttöliittymä ja karttakonfiguraatio

MIR logistiikkarobotin kartoittaminen on tehty hyvin yksinkertaiseksi. Robotti viedään paikkaan, jossa sillä halutaan operoida. Aluksi robotti käynnistetään ja yhdistetään sen omaan verkkoon, joko tabletilla tai älypuhelimella. Tämän jälkeen avataan selaimesta osoite mir.com, josta pääsee kirjautumaan suoraan MIR:in omaan käyttöliittymään.



Kuva 33. MiR100 robotti tekemässä kartoitusta Finlandia talon huoltokäytävällä

MiR100-mobiilirobotti on varustettu kahdella 270 asteen SICK LiDAR -anturilla, joita käytetään paitsi turvallisuuden valvontaan myös ympäristön havainnointiin ja kartoitukseen. Näiden antureiden avulla robotti pystyy rakentamaan kaksiulotteisen (2D) kartan ympäristöstään, mikä mahdollistaa sen paikan määrittämisen ja autonomisen liikkumisen ennalta määriteltyjen pisteiden välillä kartoitetulla alueella, kuten Kuva 34 on esitetty.

Kuva 34. Metropolia AMK Myyrmäen kampuksen lähdekerroksen kartta.

Koska kyseessä on kaksiulotteinen (2D) kartta, siihen liittyy myös x- ja y-koordinaatit. Etäisyys ” d” kahden pisteen välillä voidaan laskea seuraavalla kaavalla:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (1)$$

missä “x_2” ja “y_2” ovat toisen pisteen koordinaatit ja “x_1” ja “y_1” ensimmäisen pisteen koordinaatit. Vaikka MiR-järjestelmä pystyy automaattisesti laskemaan etäisyyksiä reitinsuunnittelun aikana, tätä laskentaa hyödynnetään myös integraatiologiikassa etäisyyteen perustuvien toimintojen määrittämiseen. Esimerkiksi, kun robotti saapuu ennalta määrätyn etäisyysrajan (5 metriä) sisälle hissin sisäänkäynnistä, voidaan hissikutsu laukaista automaattisesti odotusajan minimoimiseksi.

MiR:n käyttöliittymässä voidaan sijoittaa referenssipisteitä, joilla on tarkat x- ja y-koordinaatit kartalle. Nämä pisteet toimivat reittipisteinä tehtävien suorituksessa sekä paikkasidonnaisina triggereinä hissi-integraatiosovelluksessa. Kuten Kuva 34 esitetään, lähtökerroksen referenssipisteisiin kuuluvat nimetty aloituspiste, tuolien hakupiste, odotuspaikka hissin edessä sekä paikka hissin sisällä. Aloituspiste varmistaa, että tehtävä alkaa aina tunnetusta ja toistettavasta sijainnista, riippumatta robotin alkuperäisestä paikasta kartalla.

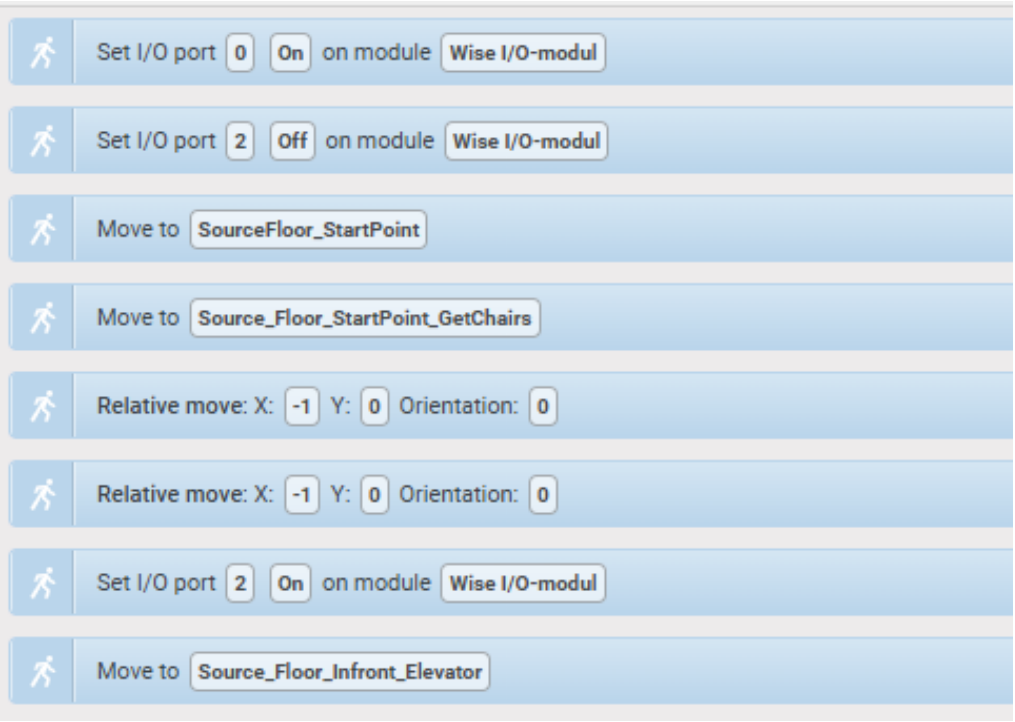
Kuva 35. Metropolia AMK Myyrmäen kampuksen kohdekerroksen kartta.

Kuva 35 esittää kohdekerroksen keskeiset referenssipisteet. Niihin sisältyvät hissin sisäinen sijainti, piste suoraan hissin ovien edessä sekä lopullinen toimituspiste, jossa kuljetettavat esineet vapautetaan. Näiden referenssipisteiden avulla varmistetaan kerrosten välinen johdonmukainen ja luotettava navigointi sekä hissin sisään- ja uloskäyntien tarkka linjaus.

6.2. Tehtävien suunnittelu ja toteutuslogiikka

Kun referenssipisteet on määritelty, luodaan tehtäväsarja, joka täyttää autonomisen logistiikkatehtävän vaatimukset. Tehtävien järjestys on suunniteltu siten, että robotti siirtyy aloituspisteestä noutopaikalle, hakee tuolit, navigoi hissin odotuspaikalle lähtökerroksessa, siirtyy hissiin sen saapuessa, matkustaa määränpäähän, poistuu hissistä ja lopulta toimittaa tuolit määrättyyn päätepisteeseen.

Suorituskyvyn yksinkertaistamiseksi ja modulaarisuuden säilyttämiseksi koko prosessi on jaettu kolmeen erilliseen tehtävään. Ensimmäinen tehtävä, joka on havainnollistettu Kuva 36, kattaa esineiden noudon ja navigoinnin hissille lähtökerroksessa. Tehtävän aikana robotti laskee nostolaitteensa Wise I/O-moduulin ohjaamana, siirtyy noutopaikalle ja asettuu siten, että nostolevy kohdistuu tarkasti tuolien kanssa. Koska nostolaite on sijoitettu robotin taakse, hyödynnetään relatiivista liikkumisstrategiaa, jonka avulla robotti peruuttaa hallitusti ja turvallisesti oikeaan asentoon.



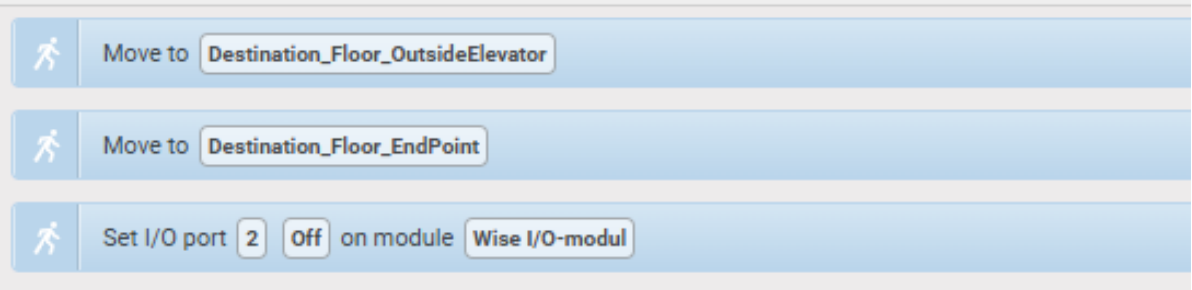
Kuva 36. Ensimmäinen tehtävä " Hissi_Integraatio_From_SourceStartPoint_to_Hissi "

Tuolien onnistuneen noston jälkeen robotti siirtyy ennalta määrättyyn odotuspaikkaan hissin ovien eteen ja jää siihen odottamaan, kunnes hissi saapuu paikalle. Tämä odotuskäyttäytyminen varmistaa, että robotti ei estä muuta liikennettä ja pitää turvallisen etäisyyden hissin sisäänkäyntiin.

Toinen tehtävä, joka on esitetty Kuva 37, ohjaa robotin kulun hissiin. Jotta poistuminen kohdekerroksessa sujuisi mahdollisimman hyvin, robotti peruuttaa hissiin siten, että tuolit ovat hissin ovia kohti. Tämä suunta mahdollistaa robotin suoran ulosajon hissistä ilman ylimääräisiä käännöksiä. Kun robotti on kokonaan hissin sisällä, aktiivinen kartta vaihdetaan kohdekerroksen kartaksi, jotta robotti voi paikantaa itsensä tarkasti kerroksen vaihdon jälkeen.

Kuva 37. Toinen Tehtävä "Source_Floor_Enter_Elevator"

Kolmas tehtävä, kuten Kuva 38 ja kartassa Kuva 35 on esitetty, käynnistyy, kun hissi saapuu kohdekerrokseen ja ovet avautuvat. Tämän tehtävän aikana robotti poistuu hissistä, navigoi lopulliseen toimituspisteeseen ja vapauttaa tuolit. Tehtävän päätyttyä autonominen toimitusjakso on valmis.



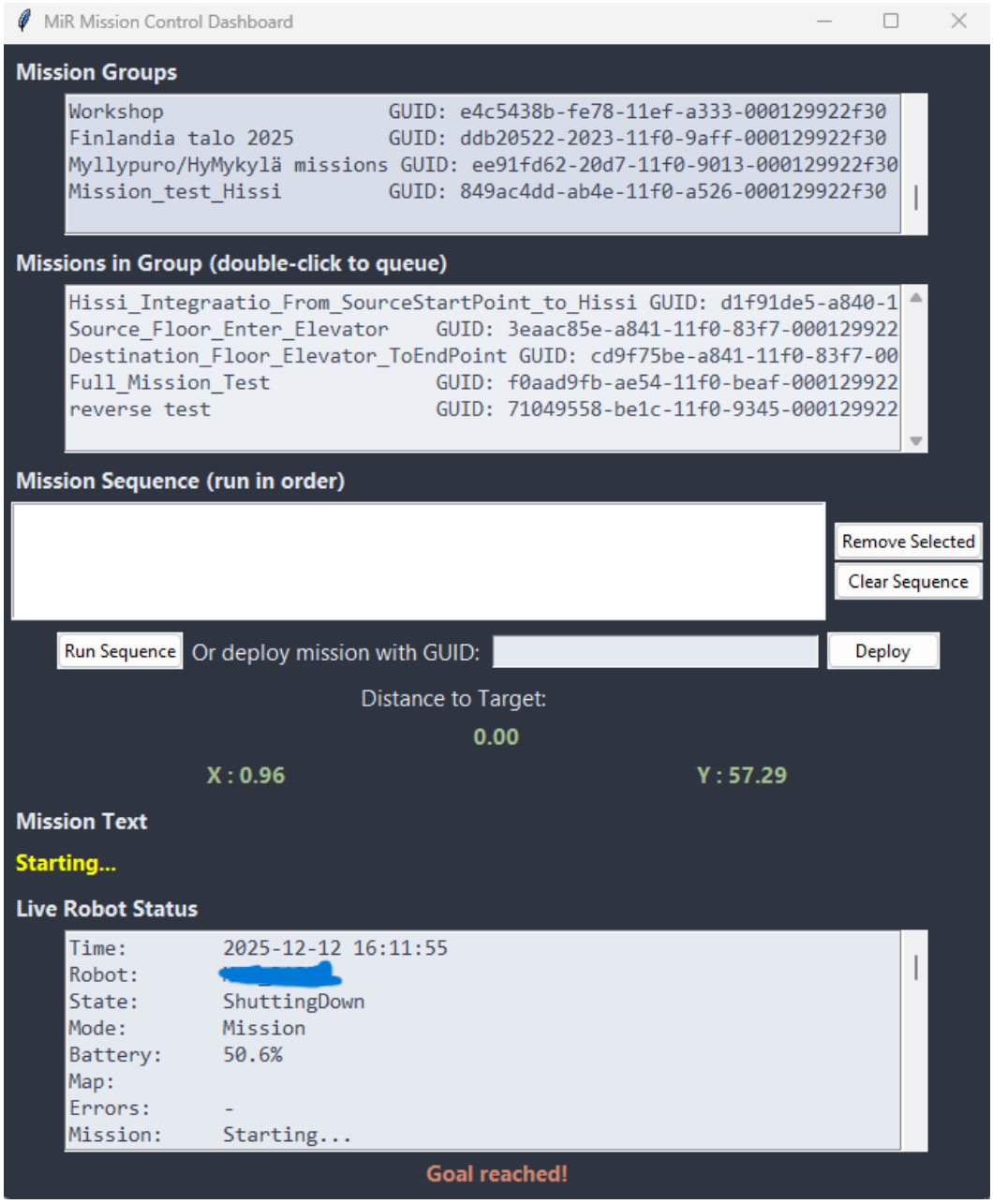
Kuva 38. Kolmas Tehtävä ” Destination_Floor_Elevator_ToEndPoint”

6.3. MiR API

Missionien suorittamisen ja monitoroinnin tueksi kehitettiin Python-pohjainen graafinen käyttöliittymä (GUI), jota kutsutaan nimellä MiR Mission Control Dashboard. Tämä käyttöliittymä, joka on esitetty Kuva 39, kommunikoi MiR-robotin kanssa sen REST API:n (versio 2.0.0) kautta ja tarjoaa operaattoreille työkalut robotin tilan visualisointiin, tehtäväryhmien selaamiseen, tehtävien jonottamiseen ja järjestettyjen tehtäväjonojen suorittamiseen.

Sovellus on toteutettu kerroksittaisen arkkitehtuurin mukaisesti ja koostuu esityskerroksesta (presentation layer), sovelluslogiikkakerroksesta (application logic layer) sekä viestintäkerroksesta (communication layer). Esityskerros on toteutettu Tkinter-frameworkilla ja se tarjoaa reaaliaikaisen näkymän tehtävien tilaan ja robotin telemetriatietoihin. Sovelluslogiikkakerros hallitsee tehtävien järjestystä, käyttäjävuorovaikutuksia ja suoritusprosessin hallintaa. Viestintäkerros vastaa REST-pohjaisesta integraatiosta MiR API:n kanssa, mukaan lukien tehtävien lähetys ja tilakyselyt.

Tämä arkkitehtoninen eriyttäminen parantaa ylläpidettävyyttä, skaalautuvuutta ja ohjelmiston vikasietoisuutta. Sovellus on toteutettu Python 3:lla ja pohjautuu yksinomaan vakiintuneisiin standardi- ja kolmannen osapuolen kirjastoihin, mikä varmistaa pitkäaikaisen ylläpidettävyyden ja toistettavuuden ISO/IEC-ohjelmistodokumentaatiokäytäntöjen mukaisesti.



Kuva 39. MiR Mission Control Dashboard Käyttöliittymä.

Sovellusarkkitehtuuri koostuu kolmesta pääkerroksesta:

- Esityskerros (GUI) – Toteutettu käyttäen tkinter-kirjastoa, ja se mahdollistaa tehtävien valinnan, järjestyksen hallinnan sekä reaaliaikaisen tilannekuvan visualisoinnin.
- Sovelluslogiikkakerros – Huolehtii tehtävien järjestyksestä, käyttäjävuorovaikutuksesta sekä suorituksen ohjauksesta.
- Viestintäkerros (MiR REST API) – Käyttää HTTP-pyyntöjä MiR-robotin kanssa tehtävien hallintaan ja tilaseurantaan.

Tämä kerroksittainen lähestymistapa selkeyttää vastuita, parantaa ylläpidettävyyttä ja laajennettavuutta. Ohjelmistoriippuvuudet määritellään ISO/IEC-dokumentaatiokäytäntöjen mukaisesti, ja jokainen ulkoinen moduuli, sen tarkoitus ja rooli järjestelmässä kuvataan eksplisiittisesti. Sovellus on toteutettu Python 3:lla ja pohjautuu yksinomaan laajasti käytettyihin ja hyvin dokumentoituihin standardi- sekä kolmannen osapuolen kirjastoihin, mikä takaa pitkäaikaisen ylläpidettävyyden sekä toistettavuuden.

Seuraava koodipätkä määrittelee tuodut riippuvuudet:

```
import tkinter as tk
from tkinter import ttk, scrolledtext, messagebox
import requests
import threading
import time
```

Näiden Python-kirjastojen roolit ovat seuraavat:

- tkinter: Vakio Python GUI -työkalu, jota käytetään Human–Machine Interface (HMI) -käyttöliittymän toteutukseen. Se mahdollistaa deterministisen tapahtumakäsittelyn, mikä sopii hyvin valvontasovelluksiin.
- ttk: Teemallinen widget-kirjasto, jolla varmistetaan yhtenäinen ja ammattimainen graafinen ilme eri käyttöjärjestelmissä.
- scrolledtext: Laajennettu widget, joka mahdollistaa vieritettävät tekstikentät – näitä käytetään esimerkiksi tehtävälistoissa ja reaaliaikaisissa tilalokeissa.
- messagebox: Tarjoaa standardoidut operaattorille tarkoitetut ilmoitukset, varoitukset ja virheikkunat.
- requests: Teollisuusstandardi HTTP-asiakaskirjasto, jota käytetään REST-pohjaiseen viestintään MiR API:n kanssa.
- threading: Mahdollistaa verkko-operaatioiden rinnakkaisen suorittamisen estämättä käyttöliittymän reagointia.
- time: Tarjoaa ajastukset, suorituksen viivästykset ja aikakatkaisujen hallinnan, jotka ovat olennaisia toimintojen ajoituksessa ja tehtäväjaksotuksessa.

The MiR-robottia ohjataan sen REST API:n kautta. Konfigurointiin tarvitaan robotin IP-osoite sekä autentikointitunnus (authentication token). Nämä parametrit on tarkoituksella ulkoistettu, jotta sovellus voidaan ottaa käyttöön eri ympäristöissä ilman, että ydinohjelmointilogiikkaa tarvitsee muokata. Tämä toteutustapa näkyy seuraavassa esimerkissä:

```
ip = ' ' # MiR robot IP address
host = f'http://{ip}/api/v2.0.0/'
headers = {
'Content-Type': 'application/json',
'Authorization': ' ' # MiR API authentication token
}
```

Tämän ansiosta sovelluksen ydintoiminnallisuudet pysyvät muuttumattomina ympäristöstä riippumatta, ja asennus voidaan tehdä helposti erilaisissa kokoonpanoissa. Keskeinen osa MiR-robotin hallintaa on mahdollisuus hakea tehtäväryhmiä ja tehtäviä (mission groups ja missions), mikä mahdollistaa joustavan tehtävien määrittelyn ja kohdistamisen. Sovellus voi hakea tehtäväryhmät MiR API:n kautta esimerkiksi seuraavalla koodilla:

```
def get_mission_groups():
    try:
        r = requests.get(host+"mission_groups", headers=headers, timeout=4)
        if r.status_code == 200:
            return r.json()
    except Exception:
        pass
    return []
def get_missions_by_group(group_guid):
    try:
        url = f"{host}mission_groups/{group_guid}/missions"
```

```
            r = requests.get(url, headers=headers, timeout=4)
            if r.status_code == 200:
                return r.json()
        except Exception:
            pass
        return []
```

Tehtävien löytäminen (mission discovery) toteutetaan funktioilla get_mission_groups() ja get_missions_by_group(). Näiden funktioiden avulla haetaan MiR-robotin muistiin tallennettua tehtäväkokoonpanodataa, jolloin käyttäjä voi dynaamisesti selailla tehtäväryhmiä ja niihin kuuluvia tehtäviä. Palautetut metatiedot, kuten tehtävien nimet ja niiden yksilölliset tunnisteet (GUIDs), esitetään graafisen käyttöliittymän yläosassa. Tämä dynaaminen haku varmistaa, että käyttöliittymä vastaa aina robotin todellista kokoonpanoa, mikä poistaa riskin vanhentuneiden tai virheellisten tehtävien suorittamisesta.

Live-käytön aikainen monitorointi toteutetaan get_status()-funktion avulla, joka hakee säännöllisin väliajoin MiR-robotin /statuspäätepisteen tiedot. Tämä päätepiste tarjoaa kattavan yhteenvedon robotin senhetkisestä tilasta, mukaan lukien navigaation eteneminen, sijaintitiedot, virran tila, tehtävän suoritus ja mahdolliset virheilmoitukset. Funktio normalisoi palautettavan tietorakenteen, jotta tiedonkäsittely on johdonmukaista, vaikka API:n vastausrakenne muuttuisi. Tällainen suojaava toteutustapa on erityisen tärkeä pitkäkestoisissa valvontajärjestelmissä, jotka toimivat todellisissa verkkoympäristöissä.

```
def get_status():
    try:
        r = requests.get(host + "status", headers=headers, timeout=4)
        if r.status_code == 200:
            d = r.json()
            if isinstance(d, list):
                d = d[0]
            return d
    except Exception:
        pass
    return None
```

get-status()-funktiosta noudetut tiedot ovat JSON-vastauksen muodossa, ja seuraavassa on esitetty, mitä tilatietoja se tarjoaa:


```
[
{
"joystick_low_speed_mode_enabled": false,
"mission_queue_url": "/v2.0.0/mission_queue/16496",
"mode_id": 7,
"moved": 232211.6226013817,
"mission_queue_id": 16496,
"robot_name": " ",
"joystick_web_session_id": "",
"uptime": 4325,
"errors": [],
"unloaded_map_changes": false,
"distance_to_next_target": 4.149810314178467,
"serial_number": "190100005001596",
"mode_key_state": "idle",
"battery_percentage": 75.9000015258789,
"map_id": " ",
```

```
"safety_system_muted": false,
"mission_text": "Moving to 'test' (4.1 meters to goal)",
"state_text": "Executing",
"velocity": {
"linear": 0.3317738175392151,
"angular": 22.2549991607666
},
"footprint": "[[-1.224,-0.13],[-1.224,0.13],[-0.454,0.32],[0.506,0.32],[0.506,-
0.32],[-0.454,-0.32]]",
"user_prompt": null,
"allowed_methods": null,
"robot_model": "MiR100",
"mode_text": "Mission",
"session_id": " ",
"state_id": 5,
"battery_time_remaining": 36442,
"position": {
"y": 8.970932006835938,
"x": 5.591537952423096,
"orientation": 148.39305114746094
}
},
{
"y": 8.970932006835938,
"x": 5.591537952423096,
"orientation": 148.39305114746094
}
]
```


Sovelluksen pääluokan poll_status()-metodi päivittää jatkuvasti haetun statustiedon graafiseen käyttöliittymään, kuten esimerkissä on osoitettu. Tämä metodi suoritetaan säännöllisin väliajoin käyttäen Tkinterin tapahtuma-ajastinta, ja toimii ensisijaisena tiedonvälityskanavana robotin tilan ja käyttöliittymän välillä.

Robotin karteesinen x- ja y-koordinaatti poimitaan JSON-vastauksesta ja näytetään selkeästi kojelaudan keskiosassa. Nämä arvot vastaavat robotin sijaintia aktiivisen kartan koordinaatistossa ja antavat operaattorille välittömän tiedon sijainnista. Navigoinnin seuraavaan kohteeseen jäljellä oleva etäisyys haetaan ja esitetään numeerisesti “Distance to Target” -kentässä, jolloin tämä parametri toimii tehtävän etenemisen avainindikaattorina. Kun etäisyys lähestyy nollaa, sovellus tulkitsee tämän tavoitteen saavuttamiseksi ja päivittää käyttöliittymän palautteella, kuten “Goal reached!”. Tämä palautemekanismi mahdollistaa intuitiivisen vahvistuksen tehtävän valmistumisesta ilman, että operaattorin tarvitsee tulkita raakatietoja.

Paikkatiedon lisäksi polling-logiikka päivittää myös tehtävän tiedon, robotin tilan, käyttömoodin, akun varaustason, aktiivisen kartan tunnisteen ja virheilmoitukset. Nämä parametrit yhdistetään rakenteiselle statusraportille, joka esitetään Live Robot Status -paneelissa. Tämä paneeli toimii valvonnan yleiskatsauksena ja mahdollistaa nopean poikkeamien, kuten matalan akun, suoritusvirheiden tai odottamattomien tilasiirtymien, tunnistamisen.

Tehtävien suorittaminen hallitaan send_mission()-funktion avulla. Tämä funktio lähettää tehtävän GUID-tunnisteen MiR:n tehtävä jonoon, kuten alla olevassa koodissa esitetään:

```
def send_mission(mission_guid):
    url = host + "mission_queue"
    data = {"mission_id": mission_guid}
    try:
        r = requests.post(url, json=data, headers=headers, timeout=4)
        return r.status_code in [200, 201]
    except Exception:
        pass
    return False
```

Kuvattu funktio toimii sekä yksittäisten tehtävien lähettämiseen että automatisoitujen tehtäväketjujen rakennuspalikkana. Paluuarvo ilmaisee yksiselitteisesti, hyväksyikö robotti tehtäväpyynnön, mikä mahdollistaa korkeamman tason logiikassa virheiden deterministisen havaitsemisen ja käsittelyn.

Tehtävien sarjallistus (mission sequencing) on yksi sovelluksen kriittisimmistä toiminnoista. Valitut tehtävät tallennetaan sisäisesti järjestettyyn listaan, joka määrittää suorituksen järjestyksen. Kun tehtäväsarja käynnistetään, tehtävät lähetetään yksi kerrallaan omassa taustasäikeessä. Kunkin tehtävän lähettämisen jälkeen järjestelmä siirtyy valvottuun odotustilaan, joka on toteutettu _wait_for_mission_done() -metodilla. Tämä metodi seuraa jatkuvasti robotin tilaa, tehtäväjonoa ja navigointietäisyyttä arvioidakseen, onko tehtävä suoritettu onnistuneesti.

Säikeistystä (threading) hyödynnetään laajasti koko sovelluksessa, jotta verkkotoiminnot ja tehtävien suorituslogiikka eivät estä graafisen käyttöliittymän vasteaikaa. Kaikki REST API -kutsut ja pitkäkestoiset tehtävät suoritetaan daemon-säikeissä, mikä varmistaa, että käyttöliittymä pysyy aina responsiivisena. Tämä suunnitteluratkaisu on erityisen tärkeä teollisissa ja tutkimusympäristöissä, joissa käyttöliittymän hidastuminen voi heikentää operaattorin luottamusta ja tilannetietoisuutta.

7. KONE Hissien API:n käyttö

KONE-hissi API:n toiminnan kokeellista validointia varten kehitettiin Python-pohjainen testiohjelma, joka kommunikoi KONE:n virtuaalisen hissiympäristön kanssa. KONE API eroaa olennaisesti perinteisistä REST-pohjaisista robottirajapinnoista, sillä se yhdistää REST-pohjaisen autentikoinnin WebSocket-pohjaiseen reaaliaikaiseen ohjaukseen ja valvontaan, mikä kuvastaa hissijärjestelmän tapahtumapohjaisuutta ja turvallisuuskriittisyyttä.

KONE tarjoaa virallisia viiterealisointeja ja esimerkkisovelluksia julkisessa GitHub-repositoriossaan osoitteessa: https://github.com/konecorp/kone-api-examples/tree/main. Tämä repository havainnollistaa palvelurobottien hissikutsujen työnkulkua KONE API:lla. Esimerkit on toteutettu pääosin TypeScriptillä, ja niitä voidaan hyödyntää arkkitehtuurisina ja loogisina ohjeina, vaikka niitä ei käytettäisi suoraan ajonaikaisina riippuvuuksina. KONE antaa myös tukea ja ohjeistusta kaikista mahdollisista komentopyynnöistä KONE API Portal -verkkosivustolla.

Projektin tässä vaiheessa virallisia esimerkkejä käytettiin viitekehyksenä viestirakenteen, sanomapohjien ja tapahtumajärjestyksen suunnitteluun, mutta varsinainen toteutus kehitettiin uudelleen Pythonilla, jotta voitiin mahdollistaa tiivis integraatio robottiohjauslogiikan kanssa ja suorittaa kokeelliset testit

virtuaalihissiympäristössä. Jotta ohjelmaa voidaan hyödyntää todellisissa sovelluksissa, KONE tarjoaa testiprosessin, jossa sovellusta testataan sekä virtuaali- että oikeissa hisseissä.

7.1. Todennus ja pääsynhallinta (Authentication and Access Control)

Pääsy KONE API:iin on suojattu OAuth 2.0 -asiakastunnisteisiin perustuvalla toiminnolla. Ennen kuin mitään hissitoimintoja voidaan suorittaa, asiakassovelluksen täytyy ensin tunnistautua KONE:n valtuutuspalvelimelle ja hankkia määräaikainen käyttöoikeustunnus (access token). Tämä tehdään HTTPS POST -pyynnöllä token-päätepisteeseen, jossa asiakastunniste, salainen avain (client secret) ja rakennuksen ID vaihdetaan käyttöoikeustunnukseen, jonka käyttöoikeudet on määritelty tarkasti. Koodi on esitetty 2.

```
CLIENT_ID = " "              # Add Client ID
CLIENT_SECRET = " "          # Add Client Secret
BUILDING_ID = "building: "   # Add Virtual Building ID
GROUP_ID = " :1"             # Add Virtual Building ID
```

Toteutettu autentikointilogiikka pyytää käyttöoikeudet (scopes), jotka liittyvät palvelurobottien hissi-API-kutsuihin sekä hissin tilavalvontaan. Onnistuneen autentikoinnin jälkeen käyttöoikeustunnuksen (access token) poimitaan JSON-vastauksesta ja liitetään suoraan WebSocket-yhteyden URL-osoitteeseen.

```
def get_access_token():
    url = "https://dev.kone.com/api/v2/oauth2/token"
    headers = {"Content-Type": "application/x-www-form-urlencoded"}
    data = {
        "grant_type": "client_credentials",
        "scope": (
            f'robotcall/group:{GROUP_ID} '
            f'application/inventory '
            f'equipmentstatus/*'
        ),
    }
    response = requests.post(url, headers=headers, data=data, auth=(CLIENT_ID,
CLIENT_SECRET))
    response.raise_for_status()

    if response.status_code == 200:
        print(f"\nConnection Response = {response.status_code}.\nAuthentication
Successful and Access Token Acquired.")
    else:
        print(f"\nAuthentication Failed! Status: {response.status_code}!\n")
        exit(1)
    return response.json()["access_token"]
```

Tämä funktio lähettää suojatun HTTPS-pyynnön, joka sisältää asiakastunnukset ja vaaditut käyttöoikeudet (scopes). Onnistuneen autentikoinnin jälkeen KONE-palvelin palauttaa JSON Web Tokenin (JWT), joka liitetään myöhemmin WebSocket-yhteyden URL-osoitteeseen. Tämä mekanismi varmistaa, että kaikki seuraavat reaaliaikaiset yhteydet KONE:n suoratoistopalveluun on autentikoitu ja valtuutettu myönnettyjen käyttöoikeuksien mukaisesti. OAuth 2.0 -protokollan käyttö vastaa alan parhaita käytäntöjä koneiden välisessä turvallisessa viestinnässä kriittisissä infrastruktuurijärjestelmissä.

Autentikoinnin jälkeen muodostetaan pysyvä WebSocket-yhteys, jonka avulla voidaan kommunikoida reaaliaikaisesti hissijärjestelmän kanssa.

```
def run_ws_client():
    token = get_access_token()
    stream_url = f"wss://dev.kone.com/stream-v2?accessToken={token}"
    ws = websocket.WebSocketApp(
        stream_url,
        on_open=on_open,
        on_message=on_message,
        on_error=on_error,
        on_close=on_close,
        subprotocols=["koneapi"]
    )
    ws.run_forever()
```

Käyttöoikeustunnuksen liittäminen suoraan WebSocket-osoitteeseen mahdollistaa sen, että KONE:n suoratoistopalvelu voi tunnistaa ja autentikoida asiakkaan heti yhteyden muodostusvaiheessa. Callback-funktioiden (on_open, on_message, on_error, on_close) käyttö mahdollistaa tapahtumapohjaisen ohjauslogiikan, mikä on olennaista, kun halutaan reagoida hissin tilan asynkronisiin muutoksiin.

7.2. Hissikutsupyyntö

Hissin kutsupyynnöt lähetetään viestimuodossa, joka noudattaa lift-call-api-v2 -spesifikaatiota. Toteutetussa ohjelmassa hissikutsu muodostetaan määrittelemällä rakennuksen tunniste (building identifier), ryhmän tunniste (group identifier), lähtöalue (source area), määränpääalue (destination area), terminaalin tunniste (terminal identifier) sekä sallittujen hissien lista. Alueet (areas) ovat loogisia kerroskoodeja, jotka on koodattu KONE:n sisäisen alueiden numerointijärjestelmän mukaisesti.

```
def make_lift_call(ws, source, destination):
    payload = {
        'type': 'lift-call-api-v2',
        'buildingId': BUILDING_ID,
        'callType': 'action',
        'groupId': "1",
        'payload': {
            'request_id': 98798789,
            'area': SOURCE_AREA,
            'time': f"{TIME}",
            'terminal': 1,
            'call': {
                'action': 2,
                'destination': DEST_AREA,
                'allowed_lifts' : [LIFT],
                }
        }
    }
    ws.send(json.dumps(payload))
    print(f"Sent lift call from floor area {source} to {destination}")
```

Kyseistä viestimuotoa kutsutaan destination payloadiksi, jossa määritellään lähtökerros (source floor) ja määränpääkerros (destination floor), eli hissille annetaan käsky ensin saapua lähtökerrokseen ja sen jälkeen siirtyä määränpäähän. Alueet (areas) edustavat loogisia kerrostunnisteita fyysisten

kerrosnumeroiden sijaan. "Allowed lifts" -kentällä ohjaus voidaan rajoittaa tiettyyn hissiautoon, mikä on erityisen hyödyllistä kontrolloiduissa testiympäristöissä.

Kun pyyntö on onnistuneesti lähetetty, hissijärjestelmä vastaa asynkronisesti kuittauksella ja kutsun tilapäivityksillä. Tämä toimintamalli vastaa todellisia palvelurobottien hissikutsuja, joissa robotti tekee kuljetuspyynnön ohjaamatta itse hissin liikettä suoraan.

KONE API mahdollistaa useita erilaisia hissikutsutyyppejä, kuten destination call, landing call, car call, transfer floor call sekä access control -pohjaiset kutsut, joista jokainen on suunniteltu erityisesti tietynlaisiin käyttötapauksiin.

“Hold_open” -pyyntö on viestipaketti, jolla voidaan pyytää pitämään hissin ovi auki tietyssä kohdassa. Näin voidaan esimerkiksi lähettää ensin destination call -pyyntö hisseille, ja ennen hissin saapumista lähtökerrokseen, lähettää “hold_open” -pyyntö, jolla varmistetaan oven auki pysyminen oikealla alueella.

```
def send_hold_open_at_source(ws):
    # Send a hold_open request to keep elevator doors open at source.
    payload = {
        "type": "lift-call-api-v2",
        "buildingId": BUILDING_ID,
        "callType": "hold_open",
        "groupId": "1",
        "payload": {
            "request_id": 132165,
            "served_area": SOURCE_AREA,
            "time": TIME,
            "lift_deck" : LIFT,
            "hard_time": 10,
            "soft_time": 30
        }
    }
    ws.send(json.dumps(payload))
    print(f"Hold Door Open Request At Source Floor ({SOURCE_FLOOR}) Has Been
Sent\n")
```

Kyllä, samaa viestipakettia (payload) voidaan käyttää lähettämällä se hissille ennen kuin se saapuu määränpääkerrokseen, jolloin hissin ovet pidetään auki 30 sekunnin ajan, kun päivitetään "served_area" määränpääalueeksi. Samalla viestirakenteella voidaan myös sulkea hissin ovet välittömästi poistamalla "soft_time" payloadista ja muuttamalla "hard_time" arvosta 10 arvoon 0.

7.3. Alueen tilaseurannan tilaus ja kutsutilan seuranta

Jotta hissin toimintaa voidaan seurata reaaliajassa, sovellus tilaa useita sivuston monitorointiaiheita (site monitoring topics) heti WebSocket-yhteyden muodostuksen jälkeen. Näihin tilauksiin sisältyvät hissin sijaintipäivitykset (nykyinen ja ilmoitettu kerros), oven tilamuutokset (OPENED, OPENING, CLOSING, CLOSED), pysähtymistapahtumat sekä hissikutsuun liittyvät kutsutilasiirtymät. Nämä tilaukset mahdollistavat tarkat tiedot siitä, missä vaiheessa hissi on, kuten lähestyykö se lähtökerrosta, avaako ovia, matkustaako määränpäähän tai onko kuljetusoperaatio valmis. Tällainen jatkuva tilaseuranta on keskeistä robotin ja hissin vuorovaikutuksen koordinoinnissa esimerkiksi monikerrosympäristöissä.

```
def subscribe_to_site_monitoring(ws, session_id):
    REQUESTID = random.randint(10000, 99999)
    SUBTOPICS = [
    f"lift_{LIFT_ID}/status",
    f"lift_{LIFT_ID}/position",
    f"lift_{LIFT_ID}/stopping",
    f"lift_{LIFT_ID}/doors",
    f'call_state/{session_id}/assigned',        #   call has been sent to lift
    f'call_state/{session_id}/fixed',           #   slowing down to source
    f'call_state/{session_id}/being_served',    #   doors opening at source
    f'call_state/{session_id}/served_soon',     #   slowing down to destination
    f'call_state/{session_id}/served',          #   doors opening at destination
    f'call_state/{session_id}/cancelled'        #    cancelled
    # Include more: action/{area}/{terminal}, call_state/session_id/fixed, etc.
as required
    ]

    payload ={
        "type": "site-monitoring",
        "requestId": f"{REQUESTID}",
        "buildingId": BUILDING_ID,
        "callType": "monitor",
        "groupId": GROUP_ID.split(":")[-1],
        "payload": {
            "sub": SUBSCRIBER_ID,
            "duration": 300,
            "subtopics": SUBTOPICS
        }
    }
    ws.send(json.dumps(payload))
    print("Subscribe Request Sent to Site Monitoring...")
```

Call state -tilaustietojen avulla saadaan yksityiskohtainen näkymä hissikutsun elinkaareen, mukaan lukien tilat kuten assigned, fixed, being_served, served_soon, served ja cancelled. Näiden tilojen seuranta mahdollistaa sen, että sovellus voi täsmällisesti tunnistaa, milloin hissi lähestyy lähtökerrosta, avaa ovet, siirtyy määränpäähän tai kuljetusoperaatio valmistuu. Tämä tieto on olennaista, jotta robotti voi ajoittaa nousemisen ja poistumisen oikea-aikaisesti monikerrosympäristöissä.

Tilausten kautta sovellus voi seurata hissin sijaintia, ovien tilaa sekä koko hissikutsun elinkaarta. Call state -tilatapahtumat tuottavat tarkan semanttisen tiedon hissin etenemisestä, mikä on kriittistä esimerkiksi robotin lähestymisen, kyytiin nousemisen tai kohdekerroksessa poistumisen oikeaan ajoitukseen.

7.4. Reaaliaikainen vastausviestien käsittely

Tuleva WebSocket-viestit käsitellään omalla viestinkäsittelijällä, joka jäsentää JSON-payloadin ja päivittää sisäiset tilamuuttujat, kuten hissin sijainnin, liikkumistilan, ovien statuksen sekä kutsun etenemisen. Hissin sijaintipäivityksiä käytetään nykyisen ja tavoitekerroksen seurantaan, kun taas ovitilaviestit ilmaisevat, ovatko ovet avautumassa, auki, sulkeutumassa vai kiinni.

```
def on_message(ws, message):
    data = json.loads(message)

    if "subtopic" in data and "data" in data:
        subtopic = data["subtopic"]
```

```
status = data["data"]

if subtopic.endswith("/position"):
    elevator_current_area = status.get("cur")
    elevator_moving = status.get("moving_state", "")

elif subtopic.endswith("/doors"):
    elevator_door_state = status.get("state")

elif subtopic.startswith("call_state/"):
    call_state = subtopic.split("/")[2]
```

Tämä logiikka muuntaa matalan tason telemetria- ja tapahtumaviestit merkityksellisiksi järjestelmätiloiksi. Esimerkiksi ovien tilapäivitysten avulla voidaan varmistaa turvalliset kyytiinnousuolosuhteet, kun taas hissikutsun tilasiirtymät ilmaisevat, milloin hissi lähestyy tai palvelee määränpääkerrosta. Koko vastaussanoman käsittelevä “on_message()”-funktio löytyy 2.

Sovellus sisältää valvontalogiikkakerroksen, joka tulkitsee nämä matalan tason tapahtumat merkityksellisiksi operatiivisiksi tiloiksi. Järjestelmä tunnistaa esimerkiksi, milloin hissi on saapunut lähtö- tai määränpääkerrokselle ja varmistaa, että ovet ovat täysin auki ennen kuin palvelun katsotaan onnistuneen. Ohjelma tukee myös valinnaisia “hold open” -komentoja oven aukioloajan pidentämiseksi, mikä havainnollistaa, miten robottijärjestelmät voivat vaikuttaa hissin toimintaan turvallisen kyytiinnousun ja purun mahdollistamiseksi.

Suorituksen aikana ohjelma tulostaa rakenteisia tilannekatsauksia, jotka sisältävät hissin sijainnin, liikkumistilan, ovien tilan ja kutsutilan siirtymät. Nämä lokit tarjoavat selkeän ajallisen tapahtumalokin hissipalveluprosessista ja toimivat todisteena dokumentoidusta API-käyttäytymisestä. Kun hissi saavuttaa määränpääkerroksen ja ovien avautuminen on vahvistettu, ohjelma päättää suorituksen hallitusti, mikä merkitsee hissikuljetusprosessin päättymistä.

Tämä kokeellinen toteutus vahvistaa, että KONE API tukee turvallista, tapahtumapohjaista hissin ohjausta, joka soveltuu integroitavaksi autonomisiin mobiilirobotteihin. WebSocket-pohjaisten tilausten käyttö mahdollistaa tarkan synkronoinnin robotin toimintojen ja hissin tilojen välillä, mihin pelkällä REST-pollauksella ei päästä.

7.5. Sovelluksen testausprosessi ja sertifiointi

Virtuaalisen hissi-integraation kehitystyön valmistuttua KONE Service Robot API:lla toteutetulle ratkaisulle suoritettiin virallinen validointi- ja testausprosessi yhteistyössä KONEen kanssa. Tämä validointivaihe oli pakollinen edellytys ratkaisun käyttöönotolle todellisissa hissiympäristöissä, ja sen tarkoituksena oli varmistaa toiminnallinen oikeellisuus, turvallisuus, vikasietoisuus ja KONEen operatiivisten sekä kyberturvallisuusstandardien noudattaminen.

Testausprosessi on KONE:n tarjoama KONE API Portalissa ja sitä ohjaa “KONE Service Robot API Solution Validation Test Guide” -dokumentti, joka sisältää kattavan joukon testitapauksia aitojen hissikäyttötilanteiden simuloimiseksi. Testit suoritettiin KONE:n tarjoamissa virtuaalihissiympäristöissä,

joiden kautta jäljiteltiin fyysistä rakennusta kohdeohjausjärjestelmineen, useine hissiryhmineen, kulunvalvontaominaisuuksineen ja häiriötilanteineen.

Validointi aloitettiin ratkaisun alustuksella ja autentikoinnin testauksella, jossa varmistettiin turvallinen API-yhteys onnistuneella OAuth-pohjaisella autentikoinnilla, resurssien löydöllä ja rakennuskonfiguraation haulla. Tämän vaiheen tarkoituksena oli varmistaa oikea organisaatiorakenne, voimassa olevat API-tunnukset ja valtuutusvastausten asianmukainen käsittely, mukaan lukien virhetilanteet. Seuraavissa testeissä keskityttiin hissin toimintamoodien tarkistamiseen, jotta varmistettiin, että robottiratkaisu tunnistaa ja reagoi oikein ei-operatiivisiin tiloihin, kuten palotila, pois käytöstä -tila, etuajo-oikeus-tila ja hissinhoitajatila. Näissä tapauksissa järjestelmän tuli estää hissikutsujen tekeminen ennakoivasti, jolloin turvallisuuskriittiset operaatiovaatimukset täyttyivät.

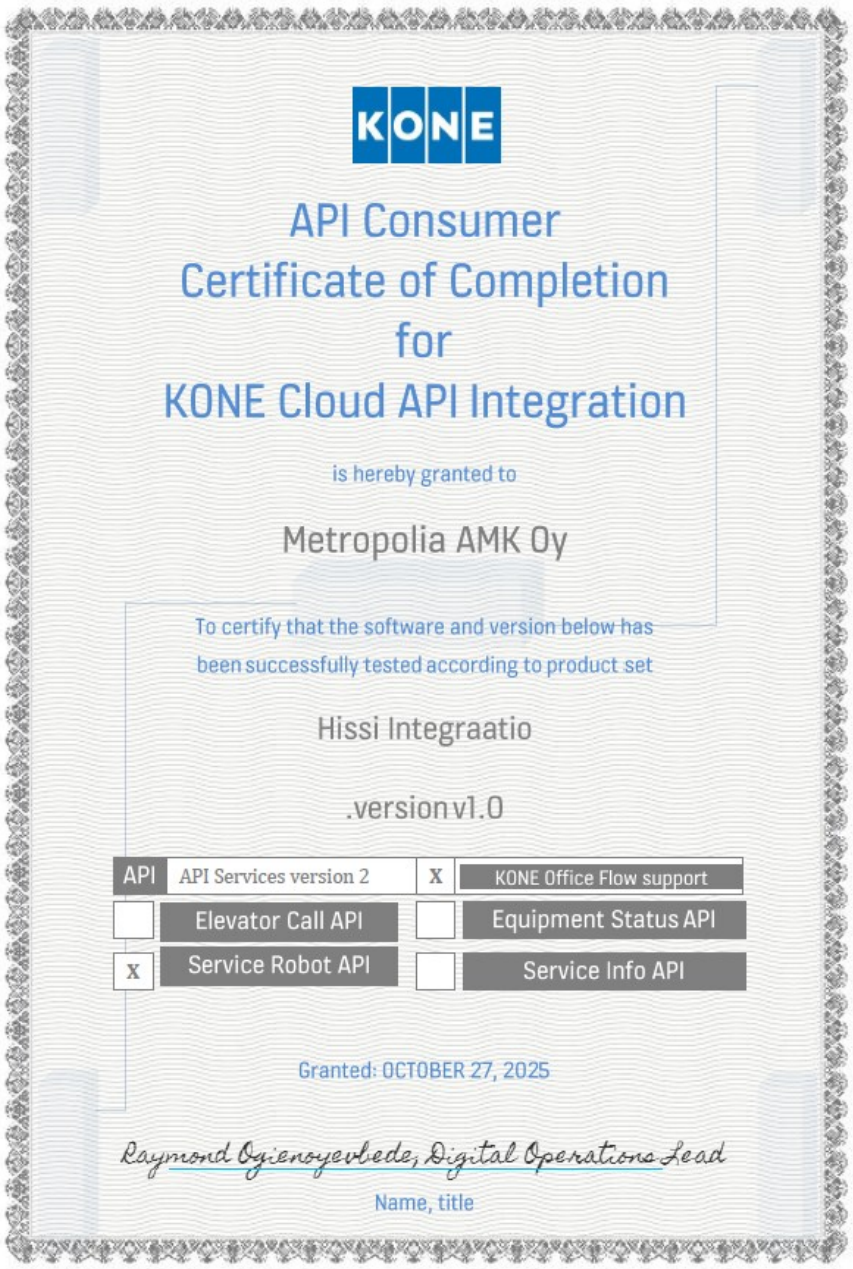
Validointi sisälsi myös hissikutsutoiminnallisuuksien testauksen laajasti: yksinkertaiset kerros- ja kabiinikutsut, viivästetyt kutsut, oven auki pitäminen, kutsun peruutus sekä vaihdolliset hissikutsut monihissijärjestelmissä. Järjestelmää testattiin myös virheellisillä ja ääritapauksilla, kuten väärillä kerrosmäärittelyillä, tukemattomilla toimintokoodeilla ja vääränlaisilla payload-tiedoilla, jotta voitiin varmistaa virheenkäsittelyn toimivuus ja järjestelmän häiriötön palautuminen ilman kokonaistoiminnan vaarantumista.

Testiohje vaati myös moniryhmähissituen validoinnin; tässä ratkaisu demonstroi oikeaa hissiryhmän tai aulan valintaa saman rakennuksen sisällä. Näin varmistettiin, että robotti pystyy itsenäisesti valitsemaan oikean hissipankin sijaintinsa ja halutun kulkusuunnan mukaan.

Kulunvalvontajärjestelmien ja sijaintiperustaisten rajoitteiden integraatio oli toinen kriittinen validointikohde. Ratkaisu toteutti onnistuneesti kerroskohtaiset käyttöoikeudet, esti kutsut luvattomista sijainneista ja varmisti, että hissikutsut sallitaan vasta fyysisten esteiden (esim. kulkuporttien) ylittämisen jälkeen. Lisäksi varmistettiin mekanismit, joilla vältettiin samanaikaisten turhien kutsujen lähettämistä odotettaessa hissin allokointia.

Kehittyneitä virhetilanteita, kuten hissin lukitusta, tilapäistä hissin poissaoloa, viestintäkatkoksia ja palautumista järjestelmän pingauksella, testattiin myös yhteistyössä KONE:n testiavustajien kanssa. Näillä skenaarioilla varmistettiin ratkaisun kestävyys, aikakatkaisujen hallinta sekä käyttäjälle annettavan palautteen toimivuus poikkeustilanteissa.

Lopuksi validointiprosessiin sisältyivät myös lokituksen, kyberturvallisuuden itsearvioinnin ja yhteydenhallinnan tarkistukset, jotta varmistettiin tapahtumien ja hissikutsujen jäljitettävyys, kyberturvavaatimusten täyttyminen sekä viestintäyhteyksien pysyvyys myös hissikabinien sisällä mahdollisesti heikentyneissä verkkoyhteyksissä.



Kuva 40. Metropolian KONE Service Robot API -sertifikaatti

Kaikki vaaditut testitapaukset läpäistiin onnistuneesti, ja ratkaisu täytti KONE:n määrittelemät hyväksymiskriteerit. Tämän seurauksena järjestelmä sai virallisen hyväksynnän ja sertifioinnin, kuten Kuva 40 esitetään, mikä oikeuttaa järjestelmän käyttöönoton ja käytön todellisissa KONE-hissiasennuksissa. Tämä sertifikaatti vahvistaa, että kehitetty ratkaisu on toiminnallisesti luotettava, turvallinen ja yhteensopiva KONE:n Service Robot API -integraatiostandardien kanssa.

8. KONEen ja MiR:n API-integraatio

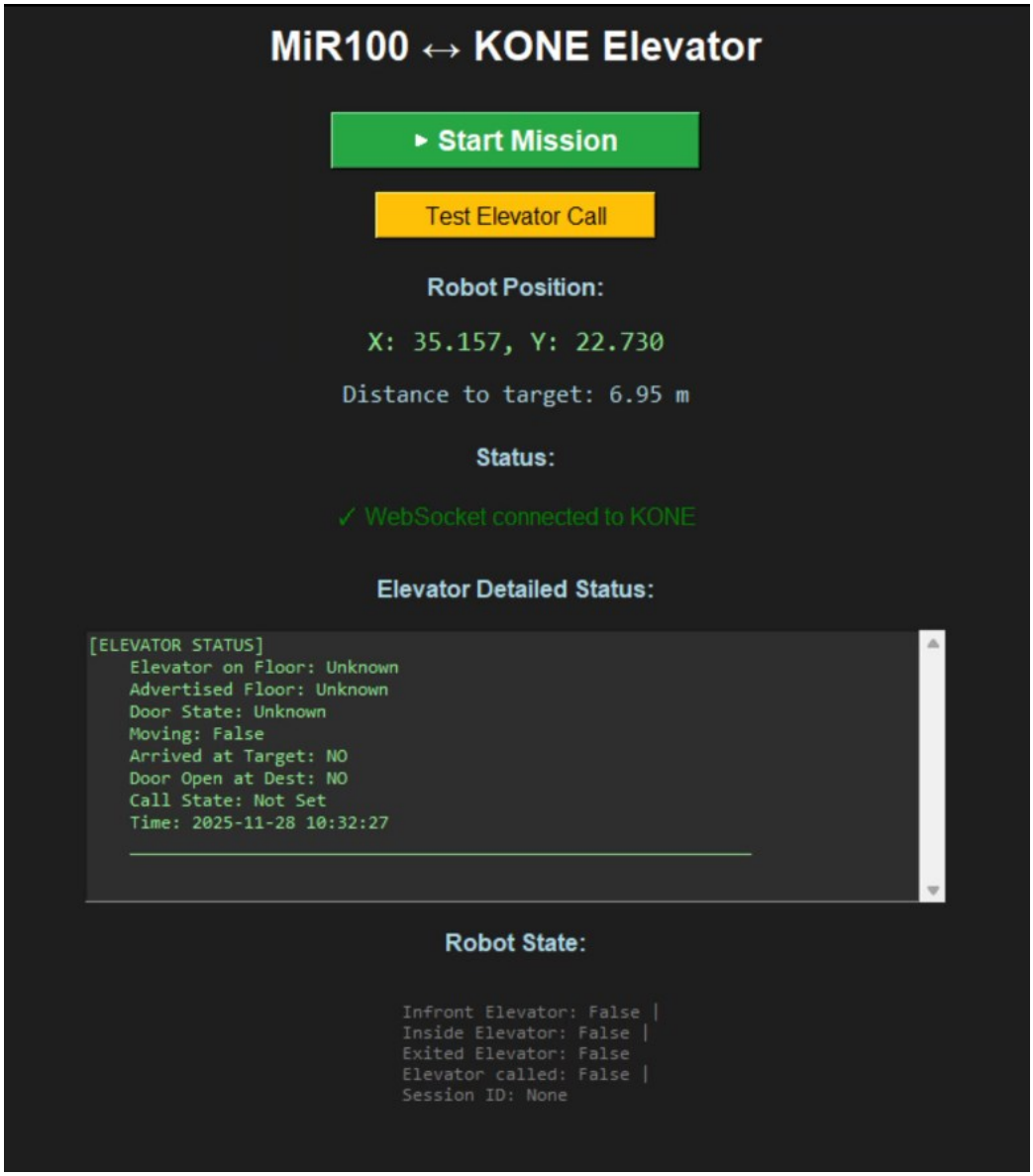
“Hissi Integraatio” on lopullinen autonominen ohjaussovellus, joka on kehitetty mahdollistamaan saumaton vuorovaikutus MiR100-mobiilirobotin ja fyysisen KONE hissijärjestelmän välillä. Sovellus yhdistää MiR REST API:n sekä KONE REST- ja WebSocket-rajapinnat yhdeksi ohjelmistoratkaisuksi, jonka avulla robotti voi liikkua itsenäisesti kerroksesta toiseen todellisessa rakennusympäristössä ilman ihmisen väliintuloa.

Järjestelmä otettiin käyttöön ja sitä testattiin oikealla MiR-mobiilirobotilla sekä kaupallisella KONE-hissiasennuksella. Kaikki hissikutsut, ovien hallintatoiminnot sekä monitorointipalautteet tulevat suoraan todellisesta hissijärjestelmästä, ja robotin paikannustiedot saadaan suoraan MiR:n navigointistackilta. Näin ollen kyseessä on aito käytäntöön perustuva ratkaisu, ei pelkkä simulaatio tai konseptimalli.

Ohjelmisto toimii valvovana kerroksena, joka koordinoi robotin navigointia, hissin käsittelyä sekä tehtävien jaksotusta molemmista järjestelmistä saatavan reaaliaikaisen palautteen perusteella. Koko lähdekoodi on nähtävissä 3. Graafinen käyttöliittymä mahdollistaa tehtävien käynnistämisen, järjestelmän tilojen visualisoinnin sekä toimintojen läpinäkyvyyden koko prosessin ajan.

8.1. GUI:n kuvaus

“Hissi Integraatio” -käyttöliittymä, joka on esitetty Kuva 41, tarjoaa hallinta- ja valvontanäkymän autonomisen robotin ja hissin väliselle vuorovaikutukselle. Käyttöliittymä on suunniteltu tukemaan reaaliaikaista käyttöä, testausta ja validointia, ei siis suoraan robotin manuaalista ohjausta.



Kuva 41. Hissi Integraatio -ohjelmiston käyttöliittymä

Sovelluksen avulla operaattori voi käynnistää täydellisen autonomisen tehtävän yhdellä ohjaustoiminnolla napsauttamalla “▶ Start Mission” -painiketta. Kun ensimmäinen tehtävä on käynnistetty, järjestelmä yhdistyy automaattisesti KONE-hissin WebSocket-rajapintaan, aloittaa MiR-robotin navigaatiotehtävän lähtökerroksessa ja käynnistää mobiilirobotin sijainnin sekä hissin tilan jatkuvan seurannan. Tehtävän aikana ei tarvita enää operaattorin lisätoimenpiteitä tai puuttumista.

Käyttöliittymä näyttää reaaliaikaiset karteesisen x- ja y-koordinaatit MiR-robotin sijainnista sekä jäljellä olevan etäisyyden seuraavaan navigaatiokohteeseen. Nämä tiedot haetaan jatkuvasti MiR REST API:sta ja ne kuvaavat mobiilirobotin todellista sijaintia rakennuksen kartalla. Arvot esitetään paitsi operaattorille, niitä käytetään myös sisäisesti hissiin liittyvien toimintojen laukaisuun.

Erillinen hissitilanäyttö esittää yksityiskohtaiset tiedot, jotka saadaan KONE:n WebSocket-pohjaisesta valvontarajapinnasta. Näihin kuuluvat muun muassa nykyinen hissin kerros, ilmoitettu määränpääkerros, ovien tila, liikkumistila, kutsun tila sekä aikaleimat. Näytetyt tiedot kuvaavat fyysisen hissin todellista tilaa ja päivittyvät jatkuvasti käytön aikana.

Käyttöliittymä havainnollistaa robotin ja hissin vuorovaikutuksen sisäistä tilaa, osoittaen onko robotti hissin edessä, hissikabiinin sisällä vai onko se poistunut määränpääkerroksessa. Tämä parantaa läpinäkyvyyttä autonomista käyttäytymistä säätelevään tilakoneeseen ja tukee vianetsintää sekä validointia aidoissa käyttötilanteissa.

8.2. Autonomisen robotin ja hissin koordinointilogiikka

Sovelluksen ydinfunktio on toteutettu jatkuvassa seurantasilmukassa, jossa yhdistetään robotin paikannustieto ja reaaliaikainen hissin tilatieto. Tämä silmukka toimii autonomisen järjestelmän päätöksenteon ytimenä ja varmistaa, että robotin toiminnot ovat synkronissa hissin käyttäytymisen kanssa todellisissa käyttöolosuhteissa.

```
def track_robot_and_elevator():
    """Main loop: Track robot position and manage elevator interaction"""
    try:
        while state.mission_started and not state.robot_exited_elevator:
            dat = get_robot_position()
            pos = dat["position"]
            dist = dat["distance"]
```

Funktio “track_robot_and_elevator()” suoritetaan jatkuvasti autonomisen tehtävän ollessa käynnissä. Jokaisella silmukan kierroksella haetaan nykyinen robotin sijainti, etäisyys seuraavaan kohdepisteeseen sekä navigoinnin eteneminen käyttäen MiR REST API:n get_robot_position()-funktiota. Näitä sijaintitietoja käytetään sekä käyttöliittymän päivitykseen että sen arvioimiseen, milloin hissin kanssa tehtävät tilasidonnaiset toiminnot tulisi käynnistää.

Kun MiR-robotti lähestyy hissiä lähtökerroksessa, sen sijaintia verrataan ennalta määriteltyyn koordinaattiin, joka edustaa hissin sisäänkäyntiä. Heti kun MiR-robotti on määritellyn etäisyysrajan sisällä (alle 5 metrin päässä lähtökerroksen hissiovista), sovellus lähettää hissikutsun KONE:n

WebSocket-rajapinnan kautta. Tämä kutsu pyytää kuljetusta lähtökerroksesta määränpääkerrokseen ja merkitsee robotin ja hissin vuorovaikutuksen alkamista.

```
if not state.elevator_called and not state.robot_inside_elevator:
                    if distance((x, y), ELEVATOR_DOOR_SOURCE_POSITION) < 5:
                        update_status("Robot at elevator - Calling elevator...",
"cyan")
                        ws_manager.send_elevator_call(SOURCE_AREA, DEST_AREA)
                        state.elevator_called = True
                        state.robot_at_elevator = True
                        state.current_mission = MISSION_ID2
```

Kun hissi saapuu lähtökerrokselle, sovellus odottaa vahvistusta siitä, että hissin ovet ovat auki — tämä perustuu reaaliaikaisiin oven tilapäivityksiin, jotka saadaan KONE:n valvontavirran kautta. Vasta kun tämä ehto täyttyy, järjestelmä laukaisee toisen MiR-tehtävän, jossa MiR-robotille annetaan käsky siirtyä hissiin. Robotin onnistunut sisäänmeno vahvistetaan sijaintipohjaisilla tarkistuksilla hissin sisäpuolella.

Kun MiR-robotti on hissin sisällä ja 0,6 metrin säteen sisällä hissin keskelle määritellystä pisteestä, se lähettää hissille pyynnön ovien sulkemiseksi. Hissin liike, kerroksen sijainti ja ovien tila arvioidaan jatkuvasti WebSocket-päivitysten avulla. Kun hissi saapuu määränpääkerrokseen ja ovet avautuvat, ohjelma käynnistää kolmannen tehtävän “MISSION_ID3”, joka ohjaa mobiilirobotin poistumaan hissistä. Ovien auki- ja sulkukäskyt annetaan tarvittaessa, jotta liikkumiselle voidaan varmistaa riittävä aika turvalliseen suoritukseen.

```
elif state.robot_inside_elevator and state.current_mission == MISSION_ID3:
                    if  not state.mission3_sent:
                        send_mir_mission(state.current_mission)
                        state.mission3_sent = True
                        update_status("\nMission ID 3 has been sent\n",
"yellow")
                        state.current_mission = None

                    if not state.robot_exited_elevator and not
state.elevator_moving and state.elevator_current_floor == DEST_FLOOR and
state.door_open_at_destination:
                        update_status("Robot exiting elevator...", "lightgreen")
                        if distance((x,y), ELEVATOR_DOOR_DESTINATION_POSITION) >
2:
                            ws_manager.send_hold_open_at_destination(CLOSEDOOR)
                            state.robot_exited_elevator = True
```

Tehtävä merkitään valmiiksi vasta, kun robotti on poistunut hissistä ja saapunut määrättyyn alueeseen määränpääkerroksessa. Tässä vaiheessa seurantasilmukka päättyy ja järjestelmä palaa lepotilaan.

9. Hissi integraation Yhteenveto

Tässä projektissa suunniteltiin, toteutettiin ja validoitiin autonominen ohjaussovellus, joka integroi MiR REST API:n sekä KONE:n REST- ja WebSocket-rajapinnat mahdollistaen MiR100-mobiilirobotin itsenäisen hissikutsujen hallinnan, ovien ohjauksen ja kerrosten välisen liikkumisen ilman ihmisen väliintuloa. Ratkaisu toimii valvovana kerroksena, joka koordinoi robotin navigointia ja hissin hallintaa.

Sovellus otettiin käyttöön ja sitä testattiin todellisessa rakennusympäristössä kaupallisen KONE-hissiasennuksen kanssa, mikä osoitti ratkaisun toimivuuden pelkän simulaation sijaan.

Virallinen validointi- ja sertifiointiprosessi toteutettiin yhteistyössä KONE:n kanssa käyttäen KONE API Portalin “KONE Service Robot API Solution Validation Test Guide” -opasta. Kaikki vaaditut testitapaukset läpäistiin menestyksekkäästi, mikä johti viralliseen hyväksyntään ja sertifiointiin oikeissa KONE-hisseissä toteutettavaa käyttöä varten. Tämä sertifikaatti vahvistaa, että ratkaisu täyttää KONE:n vaatimukset toiminnallisesta luotettavuudesta, turvallisuudesta ja vaatimustenmukaisuudesta, ja oikeuttaa ratkaisun tuotantokäyttöön todellisissa asennuksissa.

Teknisesti REST- ja WebSocket-rajapintojen yhdistetty käyttö mahdollisti viiveettömän, tapahtumapohjaisen ohjauksen ja monitoroinnin, mikä ei olisi mahdollista pelkällä REST-pollauksella. Tämä paransi hissirobotin välisen koordinaation vikasietoisuutta ja ajantasaisuutta todellisissa logistiikkatilanteissa. Nämä tulokset osoittavat, että projekti saavutti tavoitteensa: se tuotti sertifioidun, luotettavan ja turvallisen integraation, joka automatisoi rakennuksen sisäisen kuljetuksen ja tarjoaa käytännöllisen perustan skaalautuvalle älykkäälle intralogistiikalle.

Tulevaisuuden kehitystyössä tulee tutkia useiden robottien koordinaatiota monihissijärjestelmissä, rakennustason ohjausta ja liikenneoptimointia, parannettuja vikatilanteiden käsittely- ja palautumisratkaisuja, syventää kyberturvallisuutta ja valvontaa sekä standardoida käyttöönoton työkalut. Lisäksi arviointi erilaisissa rakennustyypeissä ja käyttöprofiileissa sekä kustannus–hyöty- ja läpivirtausanalyysit auttaisivat vaikutusten kvantifioinnissa ja laajemman käyttöönoton edistämisessä.

10. Projektin yhteenveto

Projektina case Finlandia talo oli ainutlaatuinen ja moniulotteinen kokonaisuus, joka tarjosi runsaasti tehtävää niin hankkeen projekti-insinööreille kuin opiskelijoillekin. Työskentely mahdollisti aidosti monialaisen oppimisen, kuten suunnittelun, mallinnuksen, valmistusprosessien, ohjelmoinnin, pintakäsittelyn ja muotoilun yhdistämisen käytännönläheiseksi kokonaisuudeksi. Lisäksi projektin yhteydessä opeteltu KONEen hissi-integrointi sekä sertifikaatti voivat luoda edellytyksiä uusille robotiikan kokeiluille Metropoliassa myöhemmin.

Projektissa syntyi viimeistelty ja toimiva autonominen robottijärjestelmä, joka siirtää tuolipinoja turvallisesti ja esteettisesti Finlandia-talon tiloissa. Ratkaisu tukee sekä käytännön logistiikkatarpeita että rakennuksen arkkitehtonista kokonaisuutta, ja se toimii esimerkkinä siitä, miten robotiikka voi palvella kulttuurihistoriallisesti arvokkaissa ympäristöissä.

Liitteet (tarvittaessa)

1. MIR Mission List and Deploy Ohjelma

Tämä Python-sovellus toteuttaa graafisen Mission Control Dashboard -käyttöliittymän MiR100-mobiilirobotille hyödyntäen MiR REST API:a. Kojelauta mahdollistaa reaaliaikaisen seurannan, tehtävien valinnan sekä tehtävien peräkkäisen suorituksen yhtenäisen käyttöliittymän kautta. Se hakee robotilta tehtäväryhmät ja ennalta määritellyt tehtävät, antaa käyttäjälle mahdollisuuden rakentaa järjestettyjä tehtäväjonoja ja varmistaa, että jokainen tehtävä suoritetaan loppuun ennen seuraavan lähettämistä.

Sovellus kysyy jatkuvasti robotin statusrajapintaa, jotta se voi esittää reaaliaikaiset telemetriatiedot, kuten sijaintikoordinaatit, etäisyyden kohteeseen, akun varauksen sekä tehtävän tilan. Taustasäikeistystä käytetään ylläpitämään käyttöliittymän responsiivisuutta verkkoviestinnän ja tehtäväsuorituksen aikana. Kokonaisuutena kojelauta toimii keskitettynä hallinta- ja valvontakomponenttina, joka tukee autonomista robottikäyttöä ja ulkoisiin rakennusjärjestelmiin integroitumista.

```
import tkinter as tk
from tkinter import ttk, scrolledtext, messagebox
import requests
import threading
import time
# ---- MiR API config ----
ip = ' ' #Add IP Address when connected to the same WiFi as MiR
host = f'http://{ip}/api/v2.0.0/'
headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Basic ' #Add MIR authentication code
}
def get_mission_groups():
    try:
        r = requests.get(host+"mission_groups", headers=headers, timeout=4)
        if r.status_code == 200:
            return r.json()
    except Exception:
        pass
    return []
def get_missions_by_group(group_guid):
    try:
        url = f"{host}mission_groups/{group_guid}/missions"
        r = requests.get(url, headers=headers, timeout=4)
        if r.status_code == 200:
            return r.json()
    except Exception:
        pass
    return []
def send_mission(mission_guid):
    url = host + "mission_queue"
    data = {"mission_id": mission_guid}
    try:
        r = requests.post(url, json=data, headers=headers, timeout=4)
        return r.status_code in [200, 201]
    except Exception:
        pass
    return False
def get_status():
    try:
```

```
            r = requests.get(host + "status", headers=headers, timeout=4)
            if r.status_code == 200:
                d = r.json()
                if isinstance(d, list):
                    d = d[0]
                return d
        except Exception:
            pass
        return None
    def mission_in_queue():
        try:
            q = requests.get(host+"mission_queue", headers=headers, timeout=4)
            if q.status_code == 200:
                queue_items = q.json()
                return len(queue_items) > 0
        except Exception:
            pass
        return False
    class MiRApp(tk.Tk):
        def __init__(self):
            super().__init__()
            self.title("MiR Mission Control Dashboard")
            self.geometry("670x780")
            self.configure(bg='#2e3440')
            fontsize = 11
            self.fontmono = ("Consolas", fontsize)
            self.fontlabel = ("Segoe UI", fontsize, "bold")
            self.fontval = ("Segoe UI", fontsize)
            pady = 5
            padx = 7
            # Mission sequence storage
            self.mission_sequence = []
            self.running_sequence = False
            # --- LAYOUT ---
            # 1. Groups
            tk.Label(self, text="Mission Groups", bg='#2e3440', fg='#e5e9f0',
    font=self.fontlabel)\
                .pack(anchor="w", pady=(pady,2), padx=padx)
            self.groups_box = scrolledtext.ScrolledText(self, width=70, height=5,
    font=self.fontmono,
                bg='#d8dee9', fg="#3b4252", insertbackground='#3b4252', bd=2,
    relief=tk.GROOVE, wrap=tk.NONE)
            self.groups_box.pack(padx=padx)
            self.groups_box.bind("<Double-Button-1>", self.on_group_doubleclick)
            # 2. Missions (can double click to add to sequence)
            tk.Label(self, text="Missions in Group (double-click to queue)",
    bg='#2e3440', fg='#e5e9f0', font=self.fontlabel)\
                .pack(anchor="w", pady=(pady,2), padx=padx)
            self.missions_box = scrolledtext.ScrolledText(self, width=70, height=6,
    font=self.fontmono,
                    bg='#eceff4', fg="#434c5e", insertbackground='#3b4252', bd=2,
    relief=tk.GROOVE, wrap=tk.NONE)
            self.missions_box.pack(padx=padx)
            self.missions_box.bind("<Double-Button-1>", self.on_mission_doubleclick)
            # 3. Mission Sequence Listbox (new)
            tk.Label(self, text="Mission Sequence (run in order)", bg='#2e3440',
    fg='#e5e9f0', font=self.fontlabel)\
                .pack(anchor="w", pady=(pady,2), padx=padx)
            seqfrm = tk.Frame(self, bg='#2e3440')
            seqfrm.pack(fill=tk.X, padx=padx, pady=(0,3))
            self.seq_listbox = tk.Listbox(seqfrm, width=60, height=4,
    font=self.fontmono)
            self.seq_listbox.pack(side=tk.LEFT, expand=1, fill=tk.BOTH)
```

```
        # Add and Remove buttons for sequence list
        btnfrm = tk.Frame(seqfrm, bg='#2e3440')
        btnfrm.pack(side=tk.LEFT, padx=(6,0))
        rembtn = ttk.Button(btnfrm, text="Remove Selected",
command=self.remove_sequence_item)
        rembtn.pack(anchor="w", pady=(2,2), fill=tk.X)
        clearbtn = ttk.Button(btnfrm, text="Clear Sequence",
command=self.clear_sequence)
        clearbtn.pack(anchor="w", pady=(0,2), fill=tk.X)
        # 4. Sequence run buttons
        seqrunfrm = tk.Frame(self, bg='#2e3440')
        seqrunfrm.pack(pady=(pady,0))
        self.seq_btn = ttk.Button(seqrunfrm, text="Run Sequence",
command=self.run_sequence)
        self.seq_btn.pack(side=tk.LEFT, padx=3)
        # individual deploy still supported
        tk.Label(seqrunfrm, text="Or deploy mission with GUID:", bg='#2e3440',
fg='#eceff4', font=self.fontval).pack(side=tk.LEFT)
        self.missionid_entry = tk.Entry(seqrunfrm, width=27, font=self.fontmono,
bg='#e5e9f0', fg='#2e3440', insertbackground='#2e3440')
        self.missionid_entry.pack(side=tk.LEFT, padx=(6,6))
        self.deploy_btn = ttk.Button(seqrunfrm, text="Deploy",
command=self.deploy_mission)
        self.deploy_btn.pack(side=tk.LEFT)
        # 5. Info grid
        infofrm = tk.Frame(self, bg='#2e3440')
        infofrm.pack(fill=tk.X, pady=(pady,0))
        tk.Label(infofrm, text="Distance to Target:", bg='#2e3440',
fg='#e5e9f0', font=self.fontval).pack(anchor="w", padx=(240,0))
        self.dist_val = tk.Label(infofrm, text="-", bg='#2e3440', fg='#a3be8c',
font=self.fontlabel)
        self.dist_val.pack(side=tk.LEFT, expand=1, fill=tk.X)
        # x/y position centered row
        xyfrm = tk.Frame(self, bg="#10234b")
        xyfrm.pack(fill=tk.X)
        self.x_val = tk.Label(xyfrm, text="X : -", bg='#2e3440', fg='#a3be8c',
font=self.fontlabel)
        self.x_val.pack(side=tk.LEFT, expand=1, fill=tk.X)
        self.y_val = tk.Label(xyfrm, text="Y : -", bg='#2e3440', fg='#a3be8c',
font=self.fontlabel)
        self.y_val.pack(side=tk.LEFT, expand=1, fill=tk.X)
        # 6. Mission Text
        tk.Label(self, text="Mission Text", bg='#2e3440', fg='#e5e9f0',
font=self.fontlabel)\
            .pack(anchor="w", pady=(pady,2), padx=padx)
        self.missiontxt_val = tk.Label(self, text="-", bg='#2e3440',
fg="#fbff00", font=("Segoe UI", fontsize, "bold"),
                                       anchor="w", width=70, wraplength=420,
justify="left")
        self.missiontxt_val.pack(anchor="w", padx=padx)
        # 7. Live Status
        tk.Label(self, text="Live Robot Status", bg='#2e3440', fg='#e5e9f0',
font=self.fontlabel)\
            .pack(anchor="w", pady=(pady,2), padx=padx)
        self.status_box = scrolledtext.ScrolledText(self, width=70, height=8,
font=self.fontmono,
                                    bg='#e5e9f0', fg='#3b4252',
insertbackground='#3b4252', bd=2, relief=tk.GROOVE, wrap=tk.WORD)
        self.status_box.pack(padx=padx)
        #Goal Status Label
        self.goal_status_label = tk.Label(self, text="", bg='#2e3440',
fg='#d08770', font=self.fontlabel)
        self.goal_status_label.pack(pady=(2,7))
```

```
        # -- Fetch groups on start
        self.after(300, self.load_groups)
        self.poll_status()
    def load_groups(self):
        self.groups_box.delete(1.0, tk.END)
        def task():
            groups = get_mission_groups()
            for g in groups:
                out = f"{g['name']:<26} GUID: {g['guid']}"
                self.groups_box.insert(tk.END, out+'\n')
        threading.Thread(target=task, daemon=True).start()
        self.missions_box.delete(1.0, tk.END)
    def on_group_doubleclick(self, event):
        self.missions_box.delete(1.0, tk.END)
        i = self.groups_box.index("@%d,%d" % (event.x, event.y))
        line = self.groups_box.get(f"{i} linestart", f"{i} lineend")
        if "GUID:" in line:
            guid = line.rsplit("GUID:", 1)[-1].strip()
            def task():
                missions = get_missions_by_group(guid)
                for m in missions:
                    out = f"{m['name']:<30} GUID: {m['guid']}"
                    self.missions_box.insert(tk.END, out+'\n')
            threading.Thread(target=task, daemon=True).start()
    def on_mission_doubleclick(self, event):
        i = self.missions_box.index("@%d,%d" % (event.x, event.y))
        line = self.missions_box.get(f"{i} linestart", f"{i} lineend")
        if "GUID:" in line:
            guid = line.rsplit("GUID:", 1)[-1].strip()
            # Add to sequence list
            self.mission_sequence.append(guid)
            self.seq_listbox.insert(tk.END, guid)
            self.status_box.insert(tk.END, f"Appended mission {guid[:8]}... to
sequence.\n")
            self.status_box.see(tk.END)
            # Also set in the single mission box for backward compat
            self.missionid_entry.delete(0, tk.END)
            self.missionid_entry.insert(0, guid)
    def remove_sequence_item(self):
        selection = self.seq_listbox.curselection()
        # Remove reverse order to avoid index shift
        for idx in reversed(selection):
            self.seq_listbox.delete(idx)
            del self.mission_sequence[idx]
    def clear_sequence(self):
        self.seq_listbox.delete(0, tk.END)
        self.mission_sequence.clear()
    def deploy_mission(self):
        # Check state before posting
        status = get_status()
        if status and status.get("state_text", "").lower() == "pause":
            messagebox.showwarning("Robot Paused", "The robot is paused. Please
press play on the robot before sending a mission.")
            self.status_box.insert(tk.END, "Robot is paused. Please press play
on the robot.\n")
            self.status_box.see(tk.END)
            return

        mission_guid = self.missionid_entry.get().strip()
        if not mission_guid:
            messagebox.showwarning("Missing", "Select or enter a mission GUID.")
            return
```

```
        threading.Thread(target=self._do_deploy, args=(mission_guid,),
daemon=True).start()
    def _do_deploy(self, mission_guid):
        ok = send_mission(mission_guid)
        ts = time.strftime("%H:%M:%S")
        msg = f"[{ts}] Mission posted [{mission_guid[:8]}...] - {'OK' if ok else
'FAILED'}\n"
        self.status_box.insert(tk.END, msg)
        self.status_box.see(tk.END)
    def run_sequence(self):
        # Check state before starting the thread
        status = get_status()
        if status and status.get("state_text", "").lower() == "pause":
            messagebox.showwarning("Robot Paused", "The robot is paused. Please
press play on the robot before starting the sequence.")
            self.status_box.insert(tk.END, "Robot is paused. Please press play
on the robot before running the sequence.\n")
            self.status_box.see(tk.END)
            return
        if self.running_sequence:
            messagebox.showwarning("Sequence Running", "A mission sequence is
already running.")
            return
        if not self.mission_sequence:
            messagebox.showwarning("No Missions", "Mission sequence is empty.")
            return
        self.running_sequence = True
        self.seq_btn.state(['disabled'])
        threading.Thread(target=self._run_sequence_thread, daemon=True).start()
    def _run_sequence_thread(self):
        seq_copy = list(self.mission_sequence) # To avoid issues with front-end
edits
        self.status_box.insert(tk.END, f"Running {len(seq_copy)}
mission(s)...\n")
        self.status_box.see(tk.END)
        for idx, mission_guid in enumerate(seq_copy):
            ok = send_mission(mission_guid)
            ts = time.strftime("%H:%M:%S")
            msg = f"[{ts}] Mission {idx+1}/{len(seq_copy)} posted
[{mission_guid[:8]}...] - {'OK' if ok else 'FAILED'}\n"
            self.status_box.insert(tk.END, msg)
            self.status_box.see(tk.END)
            if not ok:
                self.status_box.insert(tk.END, f"Failed to post mission
{mission_guid[:8]}. Aborting sequence.\n")
                break
            # Wait for completion
            done = self._wait_for_mission_done()
            if done:
                ts = time.strftime("%H:%M:%S")
                self.status_box.insert(tk.END, f"[{ts}] Mission {idx+1}
completed.\n")
                self.status_box.see(tk.END)
            else:
                self.status_box.insert(tk.END, f"Mission {idx+1} timed out or
failed. Aborting sequence.\n")
                break
        self.running_sequence = False
        self.seq_btn.state(['!disabled'])
        self.status_box.insert(tk.END, "Mission sequence done.\n")
        self.status_box.see(tk.END)
    def _wait_for_mission_done(self, max_wait=600):
```

```
        # Wait up to max_wait seconds for robot to go idle (or no mission in
queue)
        waited = 0
        interval = 1.0
        time.sleep(2.0)  # Give it time to properly report start
        while waited < max_wait:
            status = get_status()
            if status:
                state_text = status.get("state_text", "").lower()
                # Idle is safest, but some configs end mission with zero
distance
                if state_text == "idle":
                    return True
                dist = status.get("distance_to_next_target", None)
                if isinstance(dist, (int, float)) and dist == 0:
                    return True
            if not mission_in_queue():
                return True
            time.sleep(interval)
            waited += interval
        return False
    def poll_status(self):
        status = get_status()
        if status is not None:
            dist = status.get("distance_to_next_target", "-")
            pos = status.get("position", {'x':'-','y':'-'})
            xt = pos.get('x', '-')
            yt = pos.get('y', '-')
            self.dist_val.config(text=f"{dist:.2f}" if
isinstance(dist,(int,float)) else "-")
            self.x_val.config(text=f"X : {xt:.2f}" if isinstance(xt,(int,float))
else "X : -")
            self.y_val.config(text=f"Y : {yt:.2f}" if isinstance(yt,(int,float))
else "Y : -")
            mtxt = status.get("mission_text", "-")
            self.missiontxt_val.config(text=mtxt)
            state = status.get("state_text", "-")
            mode = status.get("mode_text", "-")
            bat = status.get("battery_percentage", "-")
            t_cur = time.strftime("%Y-%m-%d %H:%M:%S")
            errors = status.get('errors', [])
            if errors:
                err_strings = []
                for e in errors:
                    if isinstance(e, dict):
                        msg = e.get('message') or e.get('type') or str(e)
                    else:
                        msg = str(e)
                    err_strings.append(msg)
                errors_text = ', '.join(err_strings)
            else:
                errors_text = '-'
            try:
                if isinstance(dist, (int, float)):
                    if dist == 0:
                        self.goal_status_label.config(text="Goal reached!")
                    elif dist < 2:
                        self.goal_status_label.config(text="Mission is almost
done!")
                    else:
                        self.goal_status_label.config(text="Mission in
Progress")
                else:
```

```
                    self.goal_status_label.config(text="")
                except Exception:
                    self.goal_status_label.config(text="")
                srep = (
                    f"Time:        {t_cur}\n"
                    f"Robot:       {status.get('robot_name','-')}\n"
                    f"State:       {state}\n"
                    f"Mode:        {mode}\n"
                    f"Battery:     {bat:.1f}%\n"
                    f"Map:         {status.get('map_id','-')}\n"
                    f"Errors:      {errors_text}\n"
                    f"Mission:     {mtxt}\n"
                    "-----------------------------------\n"
                )
                self.status_box.delete('1.0', '2.0')
                self.status_box.insert('1.0', srep)
        self.after(200, self.poll_status)
if __name__ == "__main__":
    MiRApp().mainloop()
```

2. KONE API -virtuaalihissien testausohjelma

Annettu Python-skripti demonstroi automatisoitua integraatiota KONE:n hissijärjestelmään (lift) pilvi-API:n kautta robotiikka- ja ulkoisiin rakennussovelluksiin. Skripti hyödyntää websocket-client-kirjastoa ja REST API -kutsuja, autentikoituu KONE:n pilveen OAuth2-tunnuksilla, käynnistää hissikutsuja vaadituilla parametreilla (rakennus, alue, hissin ID jne.) ja tilaa reaaliaikaisia sivuston seurantatapahtumia vastaanottaakseen päivityksiä, kuten sijainti, ovien tila ja kutsun tilatiedot. Koodi mahdollistaa lisäkäskyjen lähettämisen (esim. “pidä ovi auki” lähtö- tai määränpääkerroksessa), seuraa hissin etenemistä ja hallitsee logiikkaa oikean saapumisen ja ovitoimintojen varmistamiseksi.

Suunnittelussa hyödynnetään monisäikeisiä tapahtumapohjaisia callbacks-toimintoja hissin tilamuutoksiin reagointiin, samalla kun tarjotaan säännöllistä tilapalautetta ja virheenkäsittelyä. Näin mahdollistetaan saumaton koordinaatio autonomisten järjestelmien ja älykkään hissiohjauksen välillä esimerkiksi varastorobotisaatiossa, älyrakennusten logistiikassa tai automatisoiduissa kuljetusratkaisuissa.

```
import websocket
import threading
import json
import time
import random
import requests

# ===== CONFIGURE THESE =====
CLIENT_ID = " "               # Add Client ID
CLIENT_SECRET = " "           # Add Client Secret
BUILDING_ID = "building: "    # Add Virtual Building ID
GROUP_ID = " :1"              # Add Virtual Building ID
SOURCE_AREA = 5000            # Example: Source floor (replace with your config)
DEST_AREA = 10000             # Example: Destination floor (replace with your
config)
TERMINAL = 1
LIFT_ID = 7
LIFT = 1001070
```

```
SUBSCRIBER_ID = "MIRTEST"
TIME = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
SOURCE_FLOOR = int (SOURCE_AREA/1000)
DEST_FLOOR = int (DEST_AREA/1000)

# State monitoring
received_lift_call_ack = False
elevator_current_area = None
elevator_moving = False
arrived_at_target = False
door_open_at_target = False
session_id = None
elevator_door_state = None
adv_floor = None
cur_floor = None
call_state = None
hold_open_source_sent = False
hold_open_dest_sent = False

def get_access_token():
    url = "https://dev.kone.com/api/v2/oauth2/token"
    headers = {"Content-Type": "application/x-www-form-urlencoded"}
    data = {
        "grant_type": "client_credentials",
        "scope": (
            f'robotcall/group:{GROUP_ID} '
            f'application/inventory '
            f'equipmentstatus/*'
        ),
    }
    response = requests.post(url, headers=headers, data=data, auth=(CLIENT_ID,
CLIENT_SECRET))
    response.raise_for_status()

    if response.status_code == 200:
        print(f"\nConnection Response = {response.status_code}.\nAuthentication
Successful and Access Token Acquired.")
    else:
        print(f"\nAuthentication Failed! Status: {response.status_code}!\n")
        exit(1)
    return response.json()["access_token"]

def make_lift_call(ws, source, destination):
    payload = {
        'type': 'lift-call-api-v2',
        'buildingId': BUILDING_ID,
        'callType': 'action',
        'groupId': "1",
        'payload': {
            'request_id': 98798789,
            'area': SOURCE_AREA,
            'time': f"{TIME}",
            'terminal': 1,
            'call': {
                'action': 2,
                'destination': DEST_AREA,
                'allowed_lifts' : [LIFT],
                #'delay' : 30
                #"activate_call_states" : ["assigned", "fixed", "being_served",
"served_soon", "served", "cancelled"]
                }
        }
    }
```

```
    ws.send(json.dumps(payload))
    print(f"Sent lift call from floor area {source} to {destination}")

#Not In Use Yet
def send_hold_open_at_source(ws):
    # Send a hold_open request to keep elevator doors open at source.
    payload = {
        "type": "lift-call-api-v2",
        "buildingId": BUILDING_ID,
        "callType": "hold_open",
        "groupId": "1",
        "payload": {
            "request_id": 132165,
            "served_area": SOURCE_AREA,
            "time": TIME,
            "lift_deck" : LIFT,
            "hard_time": 10,
            "soft_time": 30
        }
    }
    ws.send(json.dumps(payload))
    #print("Hold_Open Source Payload: \n" + json.dumps(payload, indent=2))
    print(f"Hold Door Open Request At Source Floor ({SOURCE_FLOOR}) Has Been
Sent\n")
    #print(f" Sent hold_open request for area {served_area} (hard={hard_time}s,
soft={soft_time}s)")

#Not In Use Yet
def send_hold_open_at_destination(ws):
    # Send a hold_open request to keep elevator doors open at source.
    payload = {
        "type": "lift-call-api-v2",
        "buildingId": BUILDING_ID,
        "callType": "hold_open",
        "groupId": "1",
        "payload": {
            "request_id": 64549876,
            "served_area": DEST_AREA,
            "time": TIME,
            "lift_deck" : LIFT,
            "hard_time": 10,
            "soft_time": 30
        }
    }
    ws.send(json.dumps(payload))
    #print("Hold_Open Destination Payload: \n" + json.dumps(payload, indent=2))
    print(f"Hold Door Open Request At Destination Floor ({DEST_FLOOR}) Has Been
Sent")
    #print(f" Sent hold_open request for area {served_area} (hard={hard_time}s,
soft={soft_time}s)")

def subscribe_to_site_monitoring(ws, session_id):
    REQUESTID = random.randint(10000, 99999)
    SUBTOPICS = [
    f"lift_{LIFT_ID}/status",
    #f"lift_{LIFT_ID}/next_stop_eta",
    f"lift_{LIFT_ID}/position",
    f"lift_{LIFT_ID}/stopping",
    f"lift_{LIFT_ID}/doors",
    f'call_state/{session_id}/assigned',              #    call has been sent
to lift
    f'call_state/{session_id}/fixed',                 #    slowing down to
source
```

```
    f'call_state/{session_id}/being_served',           #    doors opening at
source
    f'call_state/{session_id}/served_soon',            #    slowing down to
destination
    f'call_state/{session_id}/served',                 #    doors opening at
destination
    f'call_state/{session_id}/cancelled'               #    cancelled
    # Include more: action/{area}/{terminal}, call_state/session_id/fixed, etc.
as required
    ]

    payload ={
        "type": "site-monitoring",
        "requestId": f"{REQUESTID}",
        "buildingId": BUILDING_ID,
        "callType": "monitor",
        "groupId": GROUP_ID.split(":")[-1],
        "payload": {
            "sub": SUBSCRIBER_ID,
            "duration": 300,
            "subtopics": SUBTOPICS
        }
    }
    ws.send(json.dumps(payload))
    print("Subscribe Request Sent to Site Monitoring...")
    #print(f"Sent site-monitoring subscribe: {json.dumps(payload, indent=2)}")

def print_status():
    print(
        f"[STATUS] Elevator on area (floor): {elevator_current_area}, "
        f"\tDoor: {elevator_door_state},"
        f"\tMoving: {elevator_moving}, "
        f"\tArrived: {'YES' if arrived_at_target else 'NO'}, "
        f"\tDoor open at dest: {'YES' if elevator_door_state in ('OPENING',
'OPENED') else 'NO'}"
        f"\t Call State : {call_state}"
    )
    print("Time: "+ TIME +
"\n_____________________________________________________________________________
_______\n")

def on_message(ws, message):
    global received_lift_call_ack, elevator_current_area, elevator_door_state
    global elevator_moving, arrived_at_target, door_open_at_target
    global session_id, cur_floor, adv_floor, call_state
    global hold_open_source_sent, hold_open_dest_sent

    data = json.loads(message)
    #print(json.dumps(data, indent=2))

    # Capture session_id when first received
    new_session_id = data.get("data", {}).get("session_id")
    if new_session_id is not None and session_id is None:
        session_id = new_session_id
        #print(f"\n\nThe Session_ID is {session_id}\n\n")
        subscribe_to_site_monitoring(ws, session_id)

    # Handle subtopics
    if "subtopic" in data and "data" in data:
        subtopic = data["subtopic"]
        status = data["data"]

        # Lift position updates
```

```
            if subtopic.endswith("/position"):
                cur_floor = status.get("cur")
                adv_floor = status.get("adv")
                elevator_current_area = cur_floor
                elevator_moving = status.get("moving_state", "")
                if cur_floor:
                    if cur_floor == DEST_FLOOR or cur_floor == SOURCE_FLOOR:
                        #print("Lift has arrived at the required location!")
                        arrived_at_target = True
                    else:
                        arrived_at_target = False

            # Door updates
            elif subtopic.endswith("/doors"):
                elevator_door_state = status.get("state")
                if cur_floor == DEST_FLOOR and elevator_door_state == "OPENED":
                    door_open_at_target = True

            # Call state updates (assigned, fixed, being_served, served_soon,
served, cancelled)
            elif subtopic.startswith("call_state/"):
                parts = subtopic.split("/")
                if len(parts) >= 3:
                    call_state = parts[2]
                    valid_states = {"assigned", "fixed", "being_served",
"served_soon", "served", "cancelled"}
                    if call_state in valid_states:
                        #print(f"\n🚀 Call state update: {call_state}\n")
                        call_state = call_state

        if (adv_floor == cur_floor) and not hold_open_dest_sent and call_state in
["served_soon", "served"]:
            send_hold_open_at_destination(ws)
            time.sleep(2)
            if elevator_door_state == "OPENED" or "OPENING":
                hold_open_dest_sent = True
            else:
                send_hold_open_at_destination(ws)

        if (adv_floor == cur_floor) and not hold_open_source_sent and call_state in
["fixed", "being_served"]:
            send_hold_open_at_source(ws)
            time.sleep(2)
            if elevator_door_state == "OPENED" or "OPENING":
                hold_open_source_sent = True
            else:
                send_hold_open_at_source(ws)

        # Acknowledgement from lift call
        if (data.get("statusCode") == 201) and (data.get("requestId") ==
"123456789"):
            if not received_lift_call_ack:
                print("[ACK] Lift call registered.")
                received_lift_call_ack = True

        # Error handling
        if data.get("type") == "error" or data.get("statusCode", 0) >= 400:
            print(f"[ERROR] {data}")
            ws.close()

        # Print status summary every time
        print_status()
```

```
            # Optional: Exit when complete
            if arrived_at_target and door_open_at_target:
                time.sleep(30)
                print("[DONE] Elevator process complete. Exiting...")
                ws.close()

        def on_error(ws, error):
            print(f"WebSocket error: {error}")
            ws.close()

        def on_close(ws, close_status_code, close_msg):
            print(f"WebSocket closed: {close_status_code} {close_msg}")

        def on_open(ws):
            global session_id, cur_floor, adv_floor
            print("WebSocket connected. Getting initial elevator location... (wait for
        updates)")
            subscribe_to_site_monitoring(ws, session_id)
            # Wait up to 2 seconds to get initial status before calling

            def delayed_call():
                time.sleep(2)
                make_lift_call(ws, SOURCE_AREA, DEST_AREA)
            threading.Thread(target=delayed_call).start()

        def run_ws_client():
            token = get_access_token()
            stream_url = f"wss://dev.kone.com/stream-v2?accessToken={token}"
            ws = websocket.WebSocketApp(
                stream_url,
                on_open=on_open,
                on_message=on_message,
                on_error=on_error,
                on_close=on_close,
                subprotocols=["koneapi"]
            )
            ws.run_forever()

        if __name__ == "__main__":

    run_ws_client()
```

3. Hissi Integraatio Ohjelma

Tämä Python-skripti tarjoaa graafisen käyttöliittymän MiR100-autonomisen robotin ja KONE-älyhissijärjestelmän integrointiin. Tkinteriä käyttämällä skripti näyttää robotin reaaliaikaiset koordinaatit, hissin tilan ja tehtävien etenemisen, ja mahdollistaa samalla hissikutsujen manuaalisen tai automaattisen ohjauksen. Sovellus kommunikoi sekä MiR-robotin että KONE-hissin kanssa REST- ja WebSocket-rajapintojen kautta, autentikoituen molempiin järjestelmiin, seuraten hissin liikettä ja ovien tilaa reaaliajassa sekä laukaisten hissikutsun tai oven ohjauskäskyn robotin sijainnin perusteella.

Tilanhallinta ja automaattinen tapahtumien käsittely takaavat, että hissi kutsutaan ja ovia ohjataan täsmällisesti robotin lähestyessä, siirtyessä hissiin ja poistuessa sieltä, mahdollistaen täysin koordinoidun liikkumisen rakennuksessa. Virheenkäsittely, käyttöliittymän palaute ja tehtävien eteneminen tekevät tästä työkalusta käytännöllisen ratkaisun autonomisiin robotti-hissi-työnkulkuihin älyrakennuksissa.

```
import tkinter as tk
from tkinter import scrolledtext
import threading
import time
import requests
import websocket
import json
import random
import queue

# ===== MiR100 CONFIGURATION =====
MIR_IP = " " #Add IP Address when connected to the same WiFi as MiR
MIR_URL = f"http://{MIR_IP}/api/v2.0.0/"
MISSION_ID1 = " " #Add Mission ID of the mission when MiR is in the first floor
and fetches the chairs then goes to in front the elevator
MISSION_ID2 = " " #Add Mission ID of the mission when MiR enters the elevator on
the first floor and when is inside elevator changes the map of destination floor
MISSION_ID3 = " " #Add Mission ID of the mission when MiR is exiting the
elevator and goes to the end point on the destination floor and drops the
chairs.
MIR_AUTH = "Basic " #Add MIR authentication code

# ===== KONE API CONFIGURATION =====
CLIENT_ID = " " #Add Client ID
CLIENT_SECRET = " " #Add Client Secret
BUILDING_ID = "building: " #Add the building
GROUP_ID = " " #Add the GroupID building:1
TOKEN_URL = "https://dev.kone.com/api/v2/oauth2/token"
STREAM_URL = "wss://dev.kone.com/stream-v2?accessToken="

# ===== ELEVATOR CONFIGURATION =====
SOURCE_FLOOR = 1        # Source Floor
DEST_FLOOR = 3          # Destination Floor
SOURCE_AREA = SOURCE_FLOOR * 1000
DEST_AREA = DEST_FLOOR * 1000
TERMINAL = 10011
LIFT_ID = 1             # Your elevator ID
LIFT = 1001010          # Your lift deck ID
SUBSCRIBER_ID = "MIR_ROBOT_INTEGRATION"
CLOSEDOOR = 0           # For hardtime on elevator source call to close after
entering
OPENDOOR = 10           # For hardtime on elevator destination call to keep door
open in case

# ===== ROBOT SOURCE FLOOR POSITION TRIGGERS =====
ELEVATOR_DOOR_SOURCE_POSITION = (41.2, 25)  # Position where robot waits for
elevator
INSIDE_ELEVATOR_SOURCE_POSITION = (41.2, 26.3)   # Position Inside the elevator
#THRESHOLD_SORCE = 3  # Distance in meters to trigger elevator call

# ===== ROBOT DESTINATION FLOOR POSITION TRIGGERS =====
ELEVATOR_DOOR_DESTINATION_POSITION = (26, 23.45)  # Position of door where robot
exits the elevator at destination floor
INSIDE_ELEVATOR_DESTINATION_POSITION = (27, 23.45)   # Position Inside the
elevator in the Destination Floor Map
#THRESHOLD_DEST = 2  # Accuracy distance in meters

# ===== STATE MANAGEMENT =====
class RobotElevatorState:
    def __init__(self):
        # Robot states
        self.current_mission = None
        self.mission_started = False
```

```
            self.robot_at_elevator = False
            self.robot_inside_elevator = False
            self.robot_exited_elevator = False
            self.mission2_sent = False
            self.mission3_sent = False

            # Elevator call states
            self.elevator_called = False
            self.call_session_id = None
            self.call_state = None

            # Elevator position/door states
            self.elevator_current_floor = None
            self.elevator_adv_floor = None
            self.elevator_door_state = None
            self.elevator_moving = False
            self.elevator_at_source = False
            self.elevator_at_destination = False

            # Door open flags
            self.door_open_at_source = False
            self.door_open_at_destination = False

            # Hold open flags
            self.hold_open_source_sent = False
            self.hold_open_dest_sent = False

    state = RobotElevatorState()

    def update_status(msg, color="yellow"):
        """Update GUI status label"""
        status_label.config(text=msg, fg=color)
        print(f"\n[{time.strftime('%H:%M:%S')}] {msg}")

    def print_elevator_status():
        """Print detailed elevator status to GUI (like the working KONE code)"""
        # Determine if arrived at target
        arrived = "NO"
        if state.elevator_current_floor == SOURCE_FLOOR and not
    state.robot_inside_elevator:
            arrived = "YES (at SOURCE)"
        elif state.elevator_current_floor == DEST_FLOOR and
    state.robot_inside_elevator:
            arrived = "YES (at DEST)"

        # Determine if door is open at destination
        door_open_dest = "NO"
        if state.elevator_door_state in ('OPENING', 'OPENED'):
            if state.elevator_current_floor == DEST_FLOOR:
                door_open_dest = "YES"

        # Build status string
        status_text = f"""[ELEVATOR STATUS]
        Elevator on Floor: {state.elevator_current_floor if
    state.elevator_current_floor else 'Unknown'}
        Advertised Floor: {state.elevator_adv_floor if state.elevator_adv_floor else
    'Unknown'}
        Door State: {state.elevator_door_state if state.elevator_door_state else
    'Unknown'}
        Moving: {state.elevator_moving}
        Arrived at Target: {arrived}
        Door Open at Dest: {door_open_dest}
        Call State: {state.call_state if state.call_state else 'Not Set'}
```

```
    Time: {time.strftime('%Y-%m-%d %H:%M:%S')}
    ____________________________________________________________
    """

    # Update the scrolled text widget
    elevator_status_text.config(state='normal')
    elevator_status_text.delete(1.0, tk.END)
    elevator_status_text.insert(tk.END, status_text)
    elevator_status_text.config(state='disabled')
    elevator_status_text.see(tk.END)  # Auto-scroll to bottom

def get_access_token():
    """Get KONE API access token with robotcall scope"""
    headers = {"Content-Type": "application/x-www-form-urlencoded"}
    data = {
        "grant_type": "client_credentials",
        "scope": f"robotcall/group:{GROUP_ID} application/inventory
equipmentstatus/*"
    }

    try:
        response = requests.post(
            TOKEN_URL,
            headers=headers,
            data=data,
            auth=(CLIENT_ID, CLIENT_SECRET),
            timeout=10
        )

        if response.status_code == 200:
            update_status("✓ KONE Authentication successful", "green")
            return response.json()["access_token"]
        else:
            update_status(f"✗ KONE Auth failed: {response.status_code}", "red")
            print(f"Auth error: {response.text}")
            return None
    except Exception as e:
        update_status(f"✗ KONE Auth error: {e}", "red")
        return None

class KoneWebSocketManager:
    def __init__(self):
        self.ws = None
        self.token = None
        self.connected = False
        self.ws_thread = None
        self.message_queue = queue.Queue()

    def connect(self):
        """Establish WebSocket connection to KONE API"""
        if self.connected and self.ws:
            return

        try:
            self.token = get_access_token()
            if not self.token:
                update_status("Cannot connect WS - no token", "red")
                return

            ws_url = STREAM_URL + self.token

            self.ws = websocket.WebSocketApp(
                ws_url,
```

```
                    on_open=self._on_open,
                    on_message=self._on_message,
                    on_error=self._on_error,
                    on_close=self._on_close,
                    subprotocols=["koneapi"]
                )

                if self.ws_thread is None or not self.ws_thread.is_alive():
                    self.ws_thread = threading.Thread(target=self.ws.run_forever,
daemon=True)
                    self.ws_thread.start()

            except Exception as e:
                update_status(f"WebSocket connection error: {e}", "red")

        def _on_open(self, ws):
            """Handle WebSocket connection opened"""
            self.connected = True
            update_status("✓ WebSocket connected to KONE", "green")

            # Wait a moment then subscribe to monitoring
            time.sleep(0.5)

            # Process any queued messages
            while not self.message_queue.empty():
                try:
                    message = self.message_queue.get_nowait()
                    ws.send(json.dumps(message))
                except queue.Empty:
                    break

        def _on_message(self, ws, message):
            """Handle incoming WebSocket messages"""
            try:
                data = json.loads(message)

                # Print raw message to console (like working code)
                print(f"RAW WebSocket message: {message}")

                # Capture session_id when first received - FIX HERE
                new_session_id = data.get("data", {}).get("session_id")
                if new_session_id and not state.call_session_id:
                    state.call_session_id = new_session_id
                    # Convert to string before slicing
                    session_str = str(new_session_id)
                    update_status(f"Session ID received: {session_str}", "cyan")
                    self.subscribe_to_monitoring(new_session_id)

                # Handle subtopics (real-time elevator status)
                if "subtopic" in data and "data" in data:
                    self._handle_subtopic(data["subtopic"], data["data"])

                # Handle acknowledgements
                if data.get("statusCode") == 201:
                    update_status("✓ Call acknowledged", "green")
                    state.elevator_called = True

                # Handle errors
                if data.get("type") == "error" or data.get("statusCode", 0) >= 400:
                    update_status(f"✗ KONE API Error: {data}", "red")

                # Update the detailed status display
                print_elevator_status()
```

```
        except Exception as e:
            update_status(f"Message handling error: {e}", "red")
            print(f"Full error details: {e}")

    def _handle_subtopic(self, subtopic, status_data):
        """Process subtopic messages from site monitoring"""

        # Elevator position updates
        if subtopic.endswith("/position"):
            state.elevator_current_floor = status_data.get("cur")
            state.elevator_adv_floor = status_data.get("adv")
            state.elevator_moving = status_data.get("moving_state", "")

            if state.elevator_current_floor == SOURCE_FLOOR:
                state.elevator_at_source = True
                state.elevator_at_destination = False
            elif state.elevator_current_floor == DEST_FLOOR:
                state.elevator_at_destination = True
                state.elevator_at_source = False
                #send_mir_mission(MISSION_ID3)

            update_status(f"Elevator at floor {state.elevator_current_floor}",
"lightblue")

        # Door state updates
        elif subtopic.endswith("/doors"):
            state.elevator_door_state = status_data.get("state")

            if state.elevator_at_source and state.elevator_door_state in
("OPENED", "OPENING"):
                state.door_open_at_source = True
                if not state.robot_inside_elevator:
                    update_status("✓ Doors open at SOURCE - Robot can enter!",
"green")

            if state.elevator_at_destination and state.elevator_door_state in
("OPENED", "OPENING"):
                state.door_open_at_destination = True
                if state.robot_inside_elevator:
                    update_status("✓ Doors open at DESTINATION - Robot can
exit!", "green")

        # Call state updates
        elif subtopic.startswith("call_state/"):
            parts = subtopic.split("/")
            if len(parts) >= 3:
                state.call_state = parts[2]
                update_status(f"Call state: {state.call_state}", "cyan")

        #Send Hold_Open Call Logic
        if state.hold_open_dest_sent == False:
            if (state.elevator_adv_floor == state.elevator_current_floor) and
state.elevator_at_destination and state.call_state in ["served_soon", "served"]:
                time.sleep(1)
                ws_manager.send_hold_open_at_destination(OPENDOOR)
                state.hold_open_dest_sent = True

        if state.hold_open_source_sent == False:
            if (state.elevator_adv_floor == state.elevator_current_floor) and
state.elevator_at_source and state.call_state in ["fixed", "being_served"]:
                time.sleep(1)
                ws_manager.send_hold_open_at_source(OPENDOOR)
```

```
                state.hold_open_source_sent = True

    def _on_error(self, ws, error):
        """Handle WebSocket errors"""
        update_status(f"WebSocket error: {error}", "red")
        self.connected = False

    def _on_close(self, ws, close_status_code, close_msg):

        """Handle WebSocket close"""
        self.connected = False
        update_status(f"WebSocket closed: {close_status_code}", "yellow")

        # Auto-reconnect if mission is still active
        if state.mission_started and not state.robot_exited_elevator:
            threading.Timer(3.0, self.connect).start()

    def send_hold_open_at_source(self, sec):
        TIME = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
        request_id = random.randint(100000000, 999999999)
        payload = {
        "type": "lift-call-api-v2",
        "buildingId": BUILDING_ID,
        "callType": "hold_open",
        "groupId": "1",
        "payload": {
            #"request_id": request_id,
            "served_area": SOURCE_AREA,
            "time": TIME,
            "lift_deck" : LIFT,
            "hard_time": sec,
            "soft_time": 30
            }
        }
        #print(f"\n\n \t\t {payload}")
        if self.connected and self.ws:
            time.sleep(4)
            try:
                self.ws.send(json.dumps(payload))
                if sec == CLOSEDOOR:
                    update_status(f"Close Door Request At Source Floor
({SOURCE_FLOOR}) Has Been Sent", "cyan")
                else:
                    update_status(f"Hold Door Open Request At Source Floor
({SOURCE_FLOOR}) Has Been Sent", "cyan")
            except Exception as e:
                update_status(f"Send error: {e}", "red")
                self.message_queue.put(payload)
        else:
            self.message_queue.put(payload)
            update_status("Queued - connecting WebSocket...", "yellow")
            self.connect()

    def send_hold_open_at_destination(self, sec):
        TIME = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
        request_id = random.randint(100000000, 999999999)
        payload = {
        "type": "lift-call-api-v2",
        "buildingId": BUILDING_ID,
        "callType": "hold_open",
        "groupId": "1",
        "payload": {
            #"request_id": request_id,
```

```
                "served_area": DEST_AREA,
                "time": TIME,
                "lift_deck" : LIFT,
                "hard_time": sec,
                "soft_time": 30
                }
            }
            #print(f"\n\n \t\t {payload}")
            if self.connected and self.ws:
                time.sleep(4)
                try:
                    self.ws.send(json.dumps(payload))
                    if sec == CLOSEDOOR:
                        update_status(f"Close Door Request At Destination Floor
({DEST_FLOOR}) Has Been Sent", "cyan")
                    else:
                        update_status(f"Hold Door Open Request At Destination Floor
({DEST_FLOOR}) Has Been Sent", "cyan")
                except Exception as e:
                    update_status(f"Send error: {e}", "red")
                    self.message_queue.put(payload)
            else:
                self.message_queue.put(payload)
                update_status("Queued - connecting WebSocket...", "yellow")
                self.connect()

    def send_elevator_call(self, source_area, dest_area):
        """Send elevator call via WebSocket"""
        request_id = random.randint(100000000, 999999999)

        payload = {
            "type": "lift-call-api-v2",
            "buildingId": BUILDING_ID,
            "callType": "action",
            "groupId": "1",
            "payload": {
                "request_id": request_id,
                "area": source_area,
                "time": time.strftime('%Y-%m-%dT%H:%M:%SZ', time.gmtime()),
                "terminal": TERMINAL,
                "call": {
                    "action": 2,
                    "destination": dest_area,
                    "allowed_lifts": [LIFT],
                    #"delay": 30
                }
            }
        }
        if self.connected and self.ws:
            try:
                self.ws.send(json.dumps(payload))
                update_status(f"Elevator called: Floor {source_area//1000} →
{dest_area//1000}", "cyan")
            except Exception as e:
                update_status(f"Send error: {e}", "red")
                self.message_queue.put(payload)
        else:
            self.message_queue.put(payload)
            update_status("Queued - connecting WebSocket...", "yellow")
            self.connect()

    def subscribe_to_monitoring(self, session_id):
        """Subscribe to site monitoring for elevator status"""
```

```
            request_id = random.randint(100000000, 999999999)

            subtopics = [
                f"lift_{LIFT_ID}/status",
                f"lift_{LIFT_ID}/position",
                f"lift_{LIFT_ID}/stopping",
                f"lift_{LIFT_ID}/doors",
                f"call_state/{session_id}/assigned",
                f"call_state/{session_id}/fixed",
                f"call_state/{session_id}/being_served",
                f"call_state/{session_id}/served_soon",
                f"call_state/{session_id}/served",
                f"call_state/{session_id}/cancelled"
            ]

            payload = {
                "type": "site-monitoring",
                "requestId": str(request_id),
                "buildingId": BUILDING_ID,
                "callType": "monitor",
                "groupId": "1",
                "payload": {
                    "sub": SUBSCRIBER_ID,
                    "duration": 300,
                    "subtopics": subtopics
                }
            }

            if self.connected and self.ws:
                try:
                    self.ws.send(json.dumps(payload))
                    update_status("✓ Monitoring subscribed", "green")
                    print(f"Subscribed to monitoring with session_id: {session_id}")
                except Exception as e:
                    update_status(f"Subscribe error: {e}", "red")

        def disconnect(self):
            """Close WebSocket connection"""
            self.connected = False
            if self.ws:
                self.ws.close()

    # Global WebSocket manager
    ws_manager = KoneWebSocketManager()

    # ===== MiR ROBOT FUNCTIONS =====
    def start_mission(mission_id):
        """Start MiR mission and connect KONE WebSocket"""
        state.mission_started = True

        # Connect to KONE WebSocket
        ws_manager.connect()
        time.sleep(2)  # Wait for connection

        # Start MiR mission
        headers = {
            "Authorization": MIR_AUTH,
            "Content-Type": "application/json",
            "Accept-Language": "en_US"
        }
        payload = {"mission_id": mission_id}

        try:
```

```
        response = requests.post(MIR_URL + "mission_queue", headers=headers,
json=payload)
        if response.status_code == 201:
            update_status("✓ MiR Mission started", "green")
            threading.Thread(target=track_robot_and_elevator,
daemon=True).start()
        else:
            update_status(f"✗ Mission failed: {response.status_code}", "red")
    except Exception as e:
        update_status(f"✗ Mission start error: {e}", "red")

def send_mir_mission(mission_id):
    headers = {
        "Authorization": MIR_AUTH,
        "Content-Type": "application/json",
        "Accept-Language": "en_US"
    }
    payload = {"mission_id": mission_id}
    try:
        response = requests.post(MIR_URL + "mission_queue", headers=headers,
json=payload)
        if response.status_code == 201:
            update_status(f"✓ MiR Mission {mission_id} started", "green")
            return True
        else:
            update_status(f"✗ Mission {mission_id} failed:
{response.status_code}", "red")
    except Exception as e:
        update_status(f"✗ Mission {mission_id} start error: {e}", "red")
    return False

def get_robot_position():
    """Get current MiR robot position and distance to target"""
    headers = {
        "Authorization": MIR_AUTH,
        "Accept-Language": "en_US"
    }
    try:
        response = requests.get(MIR_URL + "status", headers=headers, timeout=5)
        if response.status_code == 200:
            #curentstatus = response.json()
            #print(curentstatus)
            js = response.json()
            dist = js.get("distance_to_next_target", None)
            pos = js.get("position", None)
            return {"position": pos, "distance": dist}
        else:
            update_status(f"Position API error: {response.status_code}", "red")
    except requests.RequestException as e:
        update_status(f"Position request failed: {e}", "red")
    return {"position": None, "distance": None}

def distance(p1, p2):
    """Calculate Euclidean distance between two points"""
    return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5

def track_robot_and_elevator():
    """Main loop: Track robot position and manage elevator interaction"""
    try:
        while state.mission_started and not state.robot_exited_elevator:
            dat = get_robot_position()
            pos = dat["position"]
            dist = dat["distance"]
```

```
            if pos:
                x, y = pos["x"], pos["y"]
                coord_label.config(text=f"X: {x:.3f}, Y: {y:.3f}")
                if dist is not None:
                    dist_label.config(text=f"Distance to target: {dist:.2f} m")
                else:
                    dist_label.config(text="Distance to target: -")

                # STAGE 1: Robot approaching elevator - Call elevator
                if not state.elevator_called and not
state.robot_inside_elevator:
                    if distance((x, y), ELEVATOR_DOOR_SOURCE_POSITION) < 5:
                        update_status("Robot at elevator - Calling elevator...",
"cyan")
                        ws_manager.send_elevator_call(SOURCE_AREA, DEST_AREA)
                        state.elevator_called = True
                        state.robot_at_elevator = True
                        state.current_mission = MISSION_ID2

                # STAGE 2: Wait for door to open at source, then enter
                elif state.elevator_called and state.robot_at_elevator and
state.current_mission == MISSION_ID2:
                    if not state.mission2_sent and state.door_open_at_source:
                        send_mir_mission(state.current_mission) # Enter Elevator
to Destination
                        update_status("Robot entering elevator...",
"lightgreen")
                        state.mission2_sent = True
                    state.robot_exited_elevator = False
                    if distance((x,y), INSIDE_ELEVATOR_SOURCE_POSITION) < 0.6:
                        #ws_manager.send_hold_open_at_source(CLOSEDOOR)
                        state.robot_inside_elevator = True
                        state.robot_at_elevator = False
                        state.current_mission = MISSION_ID3

                        # In real implementation, send command to robot to enter

                # STAGE 3: Robot inside - Wait for destination floor and door
open
                elif state.robot_inside_elevator and state.current_mission ==
MISSION_ID3:
                    if  not state.mission3_sent:
                        send_mir_mission(state.current_mission)
                        state.mission3_sent = True
                        update_status("\nMission ID 3 has been sent\n",
"yellow")
                        state.current_mission = None

                    if not state.robot_exited_elevator and not
state.elevator_moving and state.elevator_current_floor == DEST_FLOOR and
state.door_open_at_destination:
                        update_status("Robot exiting elevator...", "lightgreen")
                        if distance((x,y), ELEVATOR_DOOR_DESTINATION_POSITION) >
2:
                            ws_manager.send_hold_open_at_destination(CLOSEDOOR)
                            state.robot_exited_elevator = True
                            # In real implementation, send command to robot to
exit

                            update_status("✓ Mission COMPLETE - Robot exited!",
"green")
                            state.mission_started = False
```

```
                            time.sleep(1)

    except Exception as e:
        update_status(f"Tracking error: {e}", "red")
    finally:
        update_status("Mission ended", "yellow")

def manual_elevator_test():
    """Manual test: Call elevator without robot"""
    if not ws_manager.connected:
        ws_manager.connect()
        time.sleep(2)

    if ws_manager.connected:
        ws_manager.send_elevator_call(SOURCE_AREA, DEST_AREA)
        update_status("Manual elevator call sent", "cyan")
    else:
        update_status("WebSocket not connected", "red")

def on_closing():
    """Clean shutdown"""
    state.mission_started = False
    ws_manager.disconnect()
    root.destroy()

# ===== GUI =====
root = tk.Tk()
root.title("MiR-100 + KONE Elevator Integration")
root.geometry("700x800")
root.configure(bg="#1e1e1e")
root.protocol("WM_DELETE_WINDOW", on_closing)

tk.Label(
    root,
    text="MiR100 ↔ KONE Elevator",
    font=("Helvetica", 20, "bold"),
    bg="#1e1e1e",
    fg="white"
).pack(pady=15)

# Start Mission Button
start_button = tk.Button(
    root,
    text="▶ Start Mission",
    font=("Helvetica", 14, "bold"),
    bg="#28a745",
    fg="white",
    command=start_mission,
    width=20
)
start_button.pack(pady=10)

# Manual Test Button
test_button = tk.Button(
    root,
    text="Test Elevator Call",
    font=("Helvetica", 12),
    bg="#ffc107",
    fg="black",
    command=manual_elevator_test,
    width=20
)
test_button.pack(pady=5)
```

```
# Robot Coordinates
tk.Label(
    root,
    text="Robot Position:",
    font=("Helvetica", 12, "bold"),
    bg="#1e1e1e",
    fg="lightblue"
).pack(pady=(15, 5))

coord_label = tk.Label(
    root,
    text="X: 0.000, Y: 0.000",
    font=("Consolas", 14),
    bg="#1e1e1e",
    fg="lightgreen"
)
coord_label.pack(pady=5)

# Add distance label
dist_label = tk.Label(
    root,
    text="Distance to target: -",
    font=("Consolas", 13),
    bg="#1e1e1e",
    fg="lightblue"
)
dist_label.pack(pady=3)

# Status Label
tk.Label(
    root,
    text="Status:",
    font=("Helvetica", 12, "bold"),
    bg="#1e1e1e",
    fg="lightblue"
).pack(pady=(15, 5))

status_label = tk.Label(
    root,
    text="Ready - Press Start Mission",
    font=("Helvetica", 12),
    bg="#1e1e1e",
    fg="yellow",
    wraplength=650
)
status_label.pack(pady=10)

# Elevator Status Display (like print_status() in working code)
tk.Label(
    root,
    text="Elevator Detailed Status:",
    font=("Helvetica", 12, "bold"),
    bg="#1e1e1e",
    fg="lightblue"
).pack(pady=(15, 5))

elevator_status_text = scrolledtext.ScrolledText(
    root,
    width=80,
    height=12,
    font=("Consolas", 10),
    bg="#2e2e2e",
```

```
    fg="lightgreen",
    wrap=tk.WORD,
    state='disabled'
)
elevator_status_text.pack(pady=10, padx=10)

# Robot State Info
tk.Label(
    root,
    text="Robot State:",
    font=("Helvetica", 12, "bold"),
    bg="#1e1e1e",
    fg="lightblue"
).pack(pady=(5, 5))

state_info = tk.Label(
    root,
    text="",
    font=("Consolas", 9),
    bg="#1e1e1e",
    fg="gray",
    justify="left"
)
state_info.pack(pady=5)

def update_state_display():
    """Update robot state information display"""
    session_display = str(state.call_session_id) if state.call_session_id else
'None'
    info = f"""
    Infront Elevator: {state.robot_at_elevator} |
    Inside Elevator: {state.robot_inside_elevator} |
    Exited Elevator: {state.robot_exited_elevator}
    Elevator called: {state.elevator_called} |
    Session ID: {session_display}"""
    state_info.config(text=info)
    if state.mission_started:
        root.after(1000, update_state_display)

# Initialize elevator status display
print_elevator_status()

# Wrap start mission to include state display updates
original_start = start_mission
def start_with_display():
    original_start(mission_id=MISSION_ID1)
    state.current_mission = 1
    update_state_display()
start_button.config(command=start_with_display)

root.mainloop()
```