POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

Département de génie informatique et génie logiciel

INF3995

Projet de conception d'un système informatique

Documentation du projet répondant à l'appel d'offres no. H2025-INF3995 du département GIGL.

Conception d'un système d'exploration multi-robot

Équipe No ***102***

Zerouali, Amine

Gratton Fournier, Kevin Santiago

Haddadi, Issam

Hachemi Boumila, Rafik

Abassi,Yassine Mohamed Taha

Milord, Mario Junior

février 2025

1. Vue d'ensemble du projet

1.1 But du projet, porté et objectifs (Q4.1)

Le projet dans le cadre du cours INF3995 vise à concevoir, développer et démontrer une preuve de concept d'un système d'exploration multi-robot. En effet, dans l'optique d'une exploration planétaire, plusieurs robots simples et autonomes seront déployés, plutôt qu'un unique robot complexe. Ce système constitué de deux robots permet l'exploration d'une zone donnée, en communiquant entre eux avec une station de contrôle au sol, tout en transmettant les données recueillies vers une interface opérateur. Ce projet se veut une simulation réaliste d'un contrat qui pourrait être passé entre un sous-traitant et une agence spatiale dans le monde du travail. Il permettra ainsi de démontrer la faisabilité technique d'un tel système en atteignant le niveau 4 du Niveau de Maturité de la Solution (NMS), en mettant l'accent sur l'autonomie des robots, leur capacité à cartographier l'environnement, et leur interaction avec un opérateur humain.

Le système à développer comprend trois composantes principales étant une station au sol, des robots physiques et une simulation Gazebo. En ce qui concerne la station au sol, c'est une application web permettant l'interaction à l'opérateur d'interagir avec le système, d'envoyer des commandes, de surveiller l'état des robots et de visualiser les données collectées. Pour ce qui est des robots physiques, il s'agit de deux robots de type AgileX Limo [2] équipés de capteurs et finalement la simulation Gazebo est une réplique virtuelle permettant de tester les algorithmes et les fonctionnalités du système sans recourir aux robots physiques. Le projet se doit donc d'intégrer ces trois composantes de façon cohérente afin d'offrir une interface unifiée à l'opérateur.

L'objectif principal du projet est de démontrer la faisabilité d'un système d'exploration autonome composé de plusieurs robots capables de collaborer pour explorer une zone inconnue. Le système doit permettre à un opérateur de superviser les missions à distance, grâce à une interface web intuitive. Les robots devront se déplacer de façon autonome, éviter les obstacles et cartographier l'environnement en temps réel.

Biens livrables attendus:

Preliminary Design Review (PDR) - 14 février 2025

- Prototype préliminaire et présentation des premiers requis R.F.1 et R.F.2.

Critical Design Review (CDR) - 28 mars 2025

- Version intermédiaire du système.

Readiness Review (RR) - 15 avril 2025

- Version finale du système multi-robot avec tous les requis techniques.

1.2 Hypothèse et contraintes (Q3.1)

Le développement du projet jusqu'à son fonctionnement repose sur plusieurs hypothèses de base qui encadrent la conception et la mise en oeuvre du système d'exploration multi-robot:

-Les robots physiques AgileX Limo fournis seront en bon état de fonctionnement tout au long du projet.

-Les capteurs déjà installés sur les robots (IMU, lidar, etc.) seront pleinement opérationnels tout au long de la session et suffisants pour répondre aux exigences d'exploration et de cartographie.

-Le réseau WiFi mis à disposition assurera une connexion stable entre la station au sol et les robots physiques.

-Le simulateur Gazebo sera fonctionnel et permettra de reproduire fidèlement les comportements des robots physiques dans un environnement virtuel.

-L'équipe de projet aura accès aux différentes ressources pour mener à bien la conception du projet, telles que la salle de laboratoire pour les tests, les exigences techniques, etc.

-La communication au sein de l'équipe sera fluide tout au long de la session et les membres de l'équipe effectueront un travail de manière collaborative en respectant les délais et les responsabilités attribuées.

Le projet comprend plusieurs contraintes qu'elles soient techniques, humaines ou organisationnelles:

Délais fixes: Le projet est structuré autour de trois jalons incontournables : la révision préliminaire de conception (PDR) le 14 février 2025, la révision critique de conception (CDR) le 28 mars 2025, et la révision de préparation (RR) le 15 avril 2025. Ces dates imposent un rythme soutenu pour respecter les livraisons.

Charge de travail limitée : L'équipe dispose d'une charge de travail maximale de 630 heures-personnes pour l'ensemble du projet. Toute proposition dépassant ce seuil serait jugée non conforme.

Disponibilité des équipements : Les robots et le matériel fourni sont partagés entre plusieurs équipes. L'accès aux robots physiques et au local d'expérimentation est donc limité, ce qui impose une planification rigoureuse des tests sur le matériel réel.

Respect des exigences techniques : Le système doit se conformer aux exigences matérielles, logicielles, fonctionnelles et de conception spécifiées dans le document « Exigences Techniques ». Certains requis sont obligatoires et conditionnent donc l'acceptation des livrables.

Conteneurisation : Toutes les composantes logicielles, à l'exception du code embarqué sur les robots, doivent être conteneurisées avec Docker afin d'assurer la portabilité et la reproductibilité des environnements de développement et d'exécution.

Sécurité : Les robots ne peuvent être utilisés que dans les locaux désignés.

Démonstrations vidéo : Toutes les fonctionnalités du système doivent être démontrées par des vidéos claires et concises, déposées sur GitLab, pour valider les livrables.

1.3 Biens livrables du projet (Q4.1)

Preliminary Design Review (PDR) – 14 février 2025

-Documentation initiale du projet (PDF sur GitLab), présentant la planification et l'architecture préliminaire.

-Démo vidéo – Robots physiques : Réponse d'un robot à la commande "Identifier" (R.F.1).

-**Démo vidéo – Simulation Gazebo** : Exécution des commandes "Lancer la mission" et "Terminer la mission" (R.F.2).

Critical Design Review (CDR) – 28 mars 2025

-Documentation révisée du projet (PDF sur GitLab), intégrant les décisions de conception.

-Présentation technique orale (10 min).

-Démos vidéos démontrant les fonctionnalités principales (R.F.1, R.F.2, …) et la collecte de journaux.
Readiness Review (RR) – 15 avril 2025

-Documentation finale du projet (PDF sur GitLab).

-Présentation finale.

-Démos vidéos de toutes les fonctionnalités complétées.

-Document "Tests.pdf" et scripts de tests.

Suivi hebdomadaire – Chaque semaine

Rapport d'avancement sur Discord.
Mise à jour des issues et milestones sur GitLab.

2. Organisation du projet

2.1 Structure d'organisation (Q6.1)

L'équipe de projet est composée de six membres, et son fonctionnement repose sur une organisation collaborative inspirée des **principes de la méthode Agile**, plus particulièrement du **modèle Scrum**. Cette approche est privilégiée afin de favoriser la souplesse, l'adaptation rapide aux imprévus et la responsabilisation de tous les membres autour d'un objectif commun.

- Les individus et leurs interactions plutôt que les processus et outils
- Des logiciels opérationnels plutôt qu'une documentation exhaustive
- La collaboration avec les clients plutôt que la négociation contractuelle
- L'adaptation au changement plutôt que le suivi rigide d'un plan [1].

Concrètement, l'équipe travaille par **sprints de 1 mois et demi**, chacun correspondant à une série d'objectifs définis en début de cycle. Au début de chaque sprint, une **réunion de planification** est tenue pour prioriser les tâches et les répartir selon les compétences et disponibilités de chacun.

Chaque membre peut contribuer à différentes composantes du projet, mais des rôles ont été attribués pour faciliter l'organisation :

Scrum Master (Kevin Santiago Gratton Fournier) : anime les réunions quotidiennes (mêlées), facilite la communication et s'assure que les obstacles sont levés rapidement.

Product Owner (Issam Haddadi) : responsable de la vision d'ensemble du projet, du suivi des requis, et de l'interface avec les parties prenantes (professeurs, encadrants).

Équipe de développement (tous les membres) : conçoit, développe, teste et intègre les différentes composantes du système.

Les réunions d'équipe ont lieu **chaque semaine** pour faire le point sur :

- l'avancement des tâches en cours,
- les difficultés rencontrées,
- la répartition des nouvelles tâches.

Les outils de suivi sont centralisés sur GitLab : chaque fonctionnalité ou correctif fait l'objet d'une **issue**, assignée à un ou plusieurs membres, avec un suivi par **milestones** correspondant aux jalons du projet (PDR, CDR, RR). Ce suivi permet d'assurer la traçabilité et la transparence tout au long du projet.

2.2 Entente contractuelle (Q11.1)

Le type d'entente contractuelle retenu est une entente à terme. Ce choix s'explique par la nature académique du projet et les contraintes précises imposées:

Échéances fixes : Trois jalons sont imposés (PDR le 14 février, CDR le 28 mars et RR le 15 avril), nécessitant une livraison des livrables à des dates déterminées, ce qui est typique d'une entente à terme. (*Voir section 4.3 – Calendrier de projet*)

Charge de travail définie : La limite de **630 heures-personnes** impose une gestion stricte du temps consacré à chaque tâche. (*Voir section 4.1 – Estimations des coûts du projet*)

Objectif pédagogique : L'objectif principal étant d'acquérir des compétences techniques et en gestion de projet, tout en respectant des échéances précises, une entente à terme est cohérente avec ce cadre.

Cette entente assure une bonne maîtrise du calendrier et de la charge de travail, en phase avec les attentes et les exigences du cours INF3995.

3. Description de la solution

3.1 Architecture logicielle générale (Q4.5)

La figure ci-dessous présente une vue d'ensemble de l'architecture logicielle du système multi-robot. L'architecture est conçue pour être **modulaire, conteneurisée et interopérable** entre le monde physique et simulé. Elle repose sur l'intégration fluide entre **quatre composantes principales** : la station au sol, l'interface utilisateur, les robots physiques et la simulation.

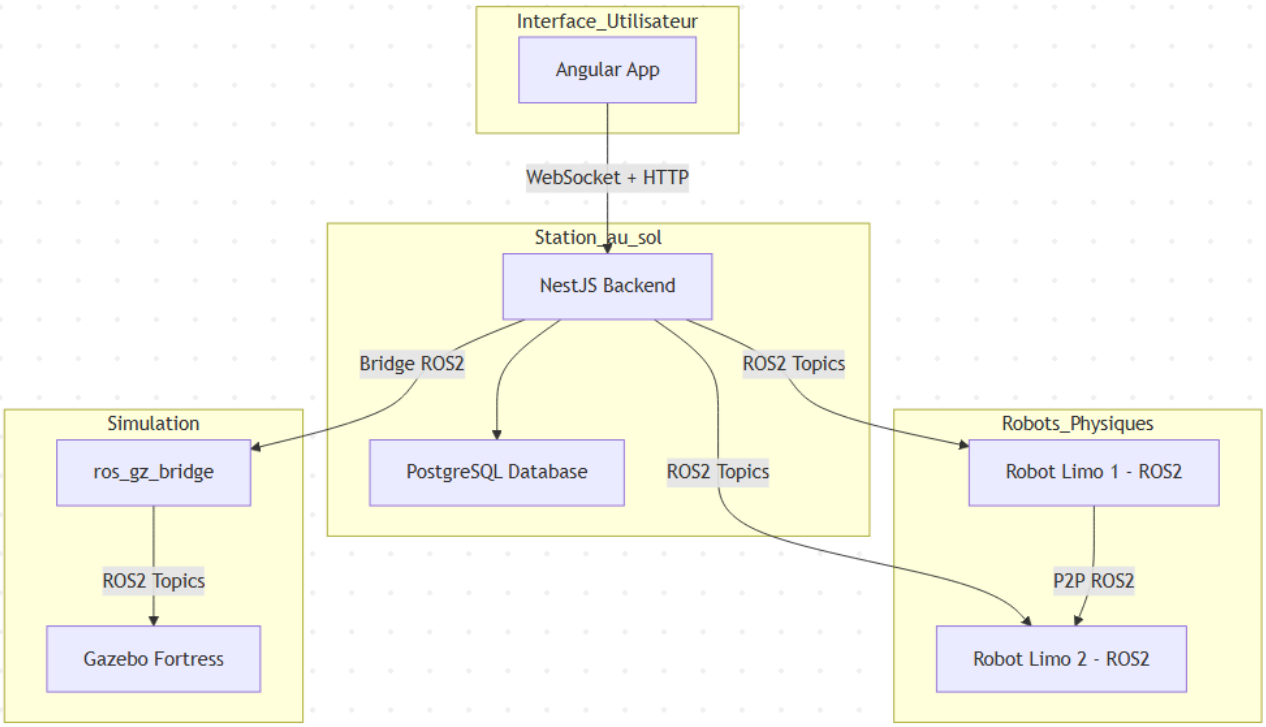


Figure 1 – Architecture logicielle générale du système

Station au sol *(NestJS, PostgreSQL)*:
R.F.2, R.F.3, R.F.8, R.C.1

La station au sol constitue le **noyau central du système**. Elle est responsable de l'orchestration des missions, de la communication bidirectionnelle avec les robots (réels ou simulés), et de la gestion de la base de données. Elle contient deux sous-composants essentiels :

- **Backend NestJS** : Fournit une API REST et WebSocket. Il reçoit les commandes issues de l'interface web, traite les données des robots, et les envoie à la base de données. Il joue aussi le rôle de relais pour les topics ROS2.
- **PostgreSQL** : Stocke les cartes générées, les journaux de missions, les positions des robots, et l'historique des commandes. Il garantit la persistance des données critiques avec robustesse.

La station au sol utilise **ROS2 Topics** pour échanger avec les robots et la simulation, et **WebSocket + HTTP** pour interagir avec l'interface utilisateur.

Interface utilisateur *(Angular)*:
R.F.10, R.C.4

L'interface utilisateur est conçue comme une **application Angular moderne** accessible sur PC, tablette ou téléphone. Elle fournit une vue complète du système :

- Lancement et arrêt des missions
- Visualisation des cartes en temps réel
- Statut et position des robots
- Journalisation des activités

Elle communique avec le backend via **Socket.IO (WebSocket)** pour les flux en temps réel et **HTTP** pour les requêtes classiques.

Logiciel embarqué *(ROS2 Humble)*:
R.F.4, R.F.5, R.F.9

Chaque robot physique exécute une instance ROS2 Humble contenant plusieurs nœuds spécialisés :

- **sensor_node** : Récupère les données des capteurs (Lidar, IMU, Caméras).
- **navigation_node** : Gère l'exploration autonome, la planification de trajectoire, et l'évitement d'obstacles.
- **communication_node** : Publie l'état du robot et reçoit les instructions de mission via ROS2 Topics.

Les robots communiquent entre eux via ROS2 en **P2P**, et avec la station via **ROS2 Topics** en Wi-Fi.

Simulation (Gazebo Fortress):
R.C.3

Gazebo Fortress permet de tester le comportement des robots dans un environnement virtuel réaliste :

- Murs, obstacles, niveaux, élévations
- Modèles de robots simulés identiques aux modèles réels
- Capteurs simulés (Lidar, IMU…)

Le pont **ros_gz_bridge** assure l'interconnexion entre la simulation et le backend ROS2.
Cela permet d'exécuter exactement les **mêmes nœuds ROS2** qu'en conditions réelles, assurant la compatibilité et facilitant les tests.

3.2 Station au sol (Q4.5)

La station au sol constitue une partie importante du système. Elle gère la communication avec les robots en temps réel, pilote les services internes de navigation et de cartographie, et assure la persistance de toutes les données critiques. Elle est conçue autour d'une **architecture modulaire** en NestJS, ce qui garantit une organisation claire, une testabilité accrue, et une évolution facilitée.

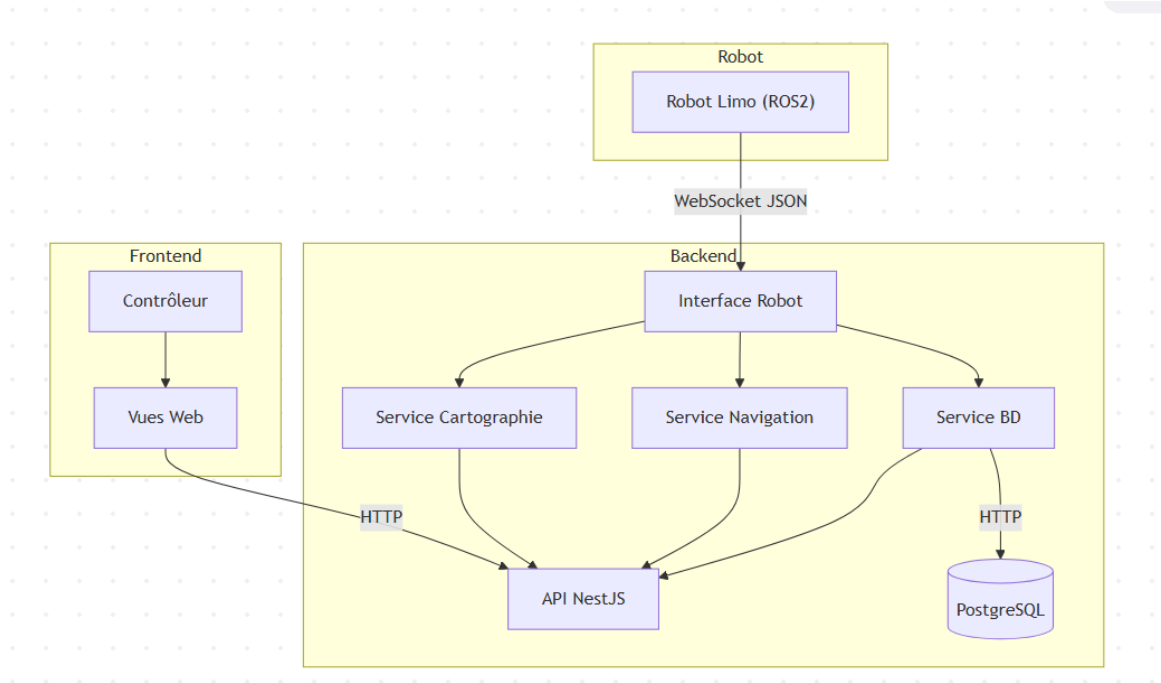


Figure 2 – Architecture logicielle de la station au sol

Architecture modulaire et séparée:

L'un des principes fondamentaux de cette architecture est la séparation claire entre la gestion des données et la communication, ce qui permet de répondre efficacement à des critères clés de robustesse et de performance temps réel.

R.F.3 : La séparation permet d'afficher les états des robots à une fréquence stable de 1 Hz, sans interférence due aux écritures en base de données.
R.F.8 : Permet une gestion fluide des cartes même en cours de mission.
R.C.1 : Assure la continuité des logs et la résilience du système en cas d'erreur réseau.

Backend (NestJS):

Interface Robot:

Ce module reçoit les messages JSON envoyés par les robots (via ROSBridge) et les distribue aux services spécialisés selon leur nature (cartographie, navigation, etc.).

Service Cartographie:

Responsable de la construction et mise à jour dynamique de la carte. Il agrège les données capteurs (lidar, IMU) et gère les cartes multiples (cartes locales par robot, carte globale fusionnée).

Service Navigation:

Gère la planification des trajectoires, le suivi de mission, et la gestion des situations spéciales (obstacle, batterie faible).

Service BD:

Ce service sert d'interface entre les composants et la base de données PostgreSQL. Il fournit des méthodes d'accès optimisées, indexées par robot, mission, timestamp, etc.

R.F.17 : Enregistrement structuré des missions (durée, positions, robots,etc.).

R.F.18 : Sauvegarde persistante des cartes

3.3 Logiciel embarqué (Q4.5)

Le logiciel embarqué est exécuté directement sur les robots physiques (LIMO), chacun équipé de **ROS2 Humble**. Il est structuré autour d'une **architecture modulaire en nœuds ROS2**, permettant d'isoler clairement les responsabilités, de garantir la fiabilité du système, et de faciliter la maintenance sur le terrain.

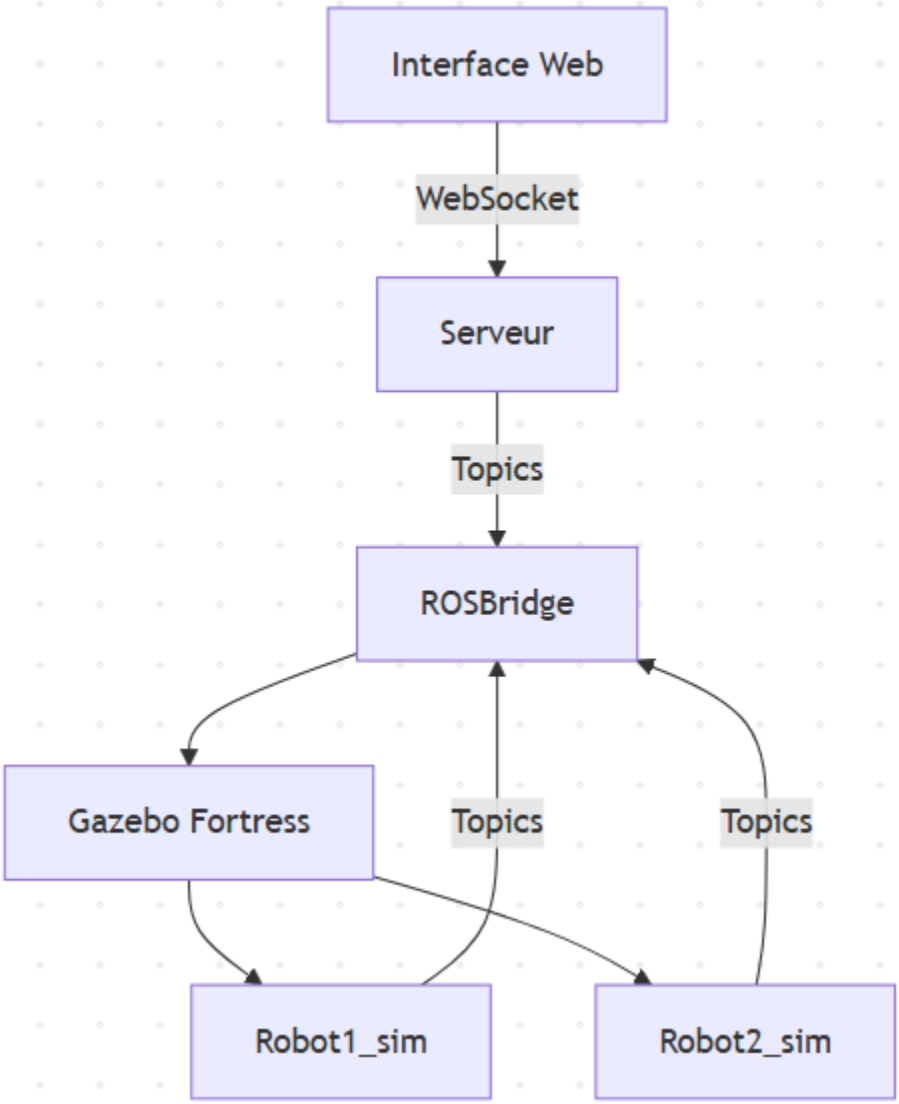


Figure 3 – Architecture logicielle embarquée sur les robots

Structure en nœuds ROS2:

ROS2 est basé sur une approche orientée composants, appelée architecture par nœuds. Chaque fonctionnalité logicielle du robot est encapsulée dans un nœud indépendant, qui communique avec les autres via des topics.

Cette approche permet :
- Une répartition claire des responsabilités
- Une réutilisabilité du code dans d'autres contextes
- Une tolérance aux pannes localisée (si un nœud échoue, les autres peuvent continuer)

Cette séparation est essentielle pour respecter les requis fonctionnels de fiabilité, d'autonomie et de robustesse (R.F.4, R.F.5, R.F.9).

Nœud communication:

Ce nœud gère la communication bidirectionnelle entre le robot et la station au sol, via WebSocket JSON encapsulé par ROSBridge. Il est responsable de :
- Publier l'état du robot (position, batterie, statut de mission)
- Recevoir les commandes de mission
- Gérer les connexions réseau et assurer la reconnexion automatique si nécessaire

Ce nœud permet d'isoler la logique réseau, évitant de polluer les modules critiques avec des aspects de connectivité.

Nœud capteurs:

Le rôle de ce nœud est d'interfacer les capteurs embarqués du robot :

- Lidar 2D/3D : Pour la détection d'obstacles
- IMU : Pour l'orientation et la stabilisation
- Caméras RGB ou RGBD : Pour la perception visuelle de l'environnement

Nœud navigation:
Cest ce qui permet aux robots de naviguer de façon autonome. Il est responsable de :
- Lire les objectifs de mission depuis un topic
- Planifier des trajectoires adaptées avec des algorithmes
- Éviter les obstacles en temps réel
- Mettre à jour l'état de progression de la mission

Ce nœud peut être configuré pour réagir à des événements critiques (obstacle bloquant, batterie faible), ce qui permet au robot d'agir **sans intervention de la station**.

3.4 Simulation (Q4.5)

La simulation est un pilier essentiel du système, car elle permet de **tester les comportements robotiques, les algorithmes de navigation et les échanges de données** avant de les déployer sur les robots physiques. Elle contribue à une meilleure fiabilité globale du système, tout en réduisant les risques de dysfonctionnement sur le terrain.

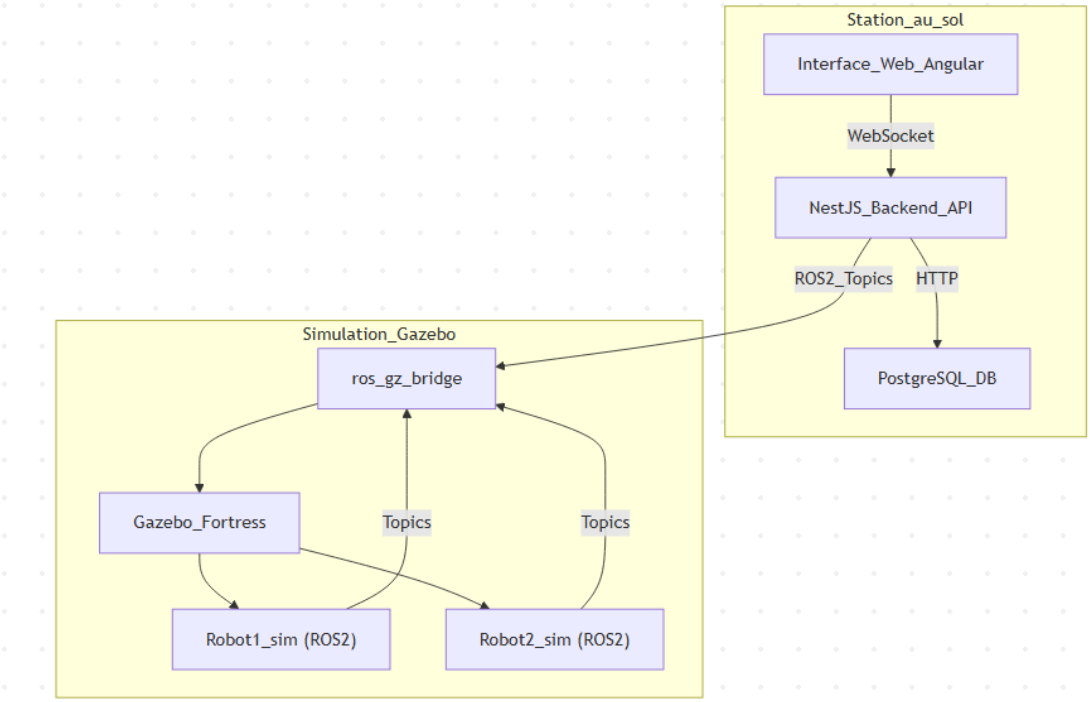


Figure 4 – Architecture de la simulation avec Gazebo, ros_gz_bridge et ROS2

Gazebo Fortress:

Gazebo Fortress est utilisé pour reproduire un environnement réaliste, comportant :

- Des obstacles (murs, objets)
- Un sol irrégulier
- Des robots Limo simulés (Robot1_sim et Robot2_sim)

Chaque robot virtuel dispose de capteurs simulés (Lidar, IMU, caméras) et peut répondre à des commandes de mouvement comme un robot réel.

Cela permet de simuler :

- La navigation autonome (R.F.4)
- La détection et l'évitement d'obstacles (R.F.5)
- Les situations critiques (batterie faible, robot bloqué, etc.)

ros_gz_bridge:

Le pont ros_gz_bridge est un élément clé de l'architecture de simulation. Il permet aux composants ROS2 de publier et de s'abonner aux topics générés par Gazebo, comme si les robots simulés étaient réels.

Par exemple :

- `/cmd_vel` permet de commander le déplacement du robot.
- `/scan` reçoit les données du lidar.
- `/odom` donne la position du robot.

Grâce à cette passerelle, **le même code ROS2 utilisé en production sur les robots physiques est également utilisé dans la simulation**, garantissant une **compatibilité totale** (R.C.3).

Robots simulés (Robot1_sim, Robot2_sim):

Ces entités exécutent des nœuds ROS2 dans un contexte simulé. Le comportement est identique à celui des robots physiques:

- Exécution des mêmes nœuds (`identify_node`, `mission_node`, `communication_node`)
- Réception de commandes depuis la station
- Envoi d'informations en temps réel via topics

Ils interagissent directement avec l'environnement de Gazebo (via capteurs) et avec la station au sol via ROS2 Topics.

Intégration avec la station au sol:

La **station au sol** (backend NestJS) est connectée à la simulation de la même manière qu'aux robots physiques :

- Envoie les commandes via **ROS2 Topics**
- Reçoit les données en temps réel (statut, cartes, positions)
- Peut lancer des missions depuis l'**interface Angular** comme si les robots étaient réels

3.5 Interface utilisateur (Q4.6)

L'interface utilisateur joue un rôle crucial dans l'exploitation du système : elle permet à l'opérateur de **lancer des missions**, **superviser les robots en temps réel**, **consulter les cartes générées**, et **réagir aux événements** (perte de robot, détection d'obstacles, etc.).

Elle est entièrement développée avec **Angular**, un framework frontend moderne, modulaire et maintenable. L'architecture suit une **séparation claire des responsabilités** via des composants Angular spécialisés.

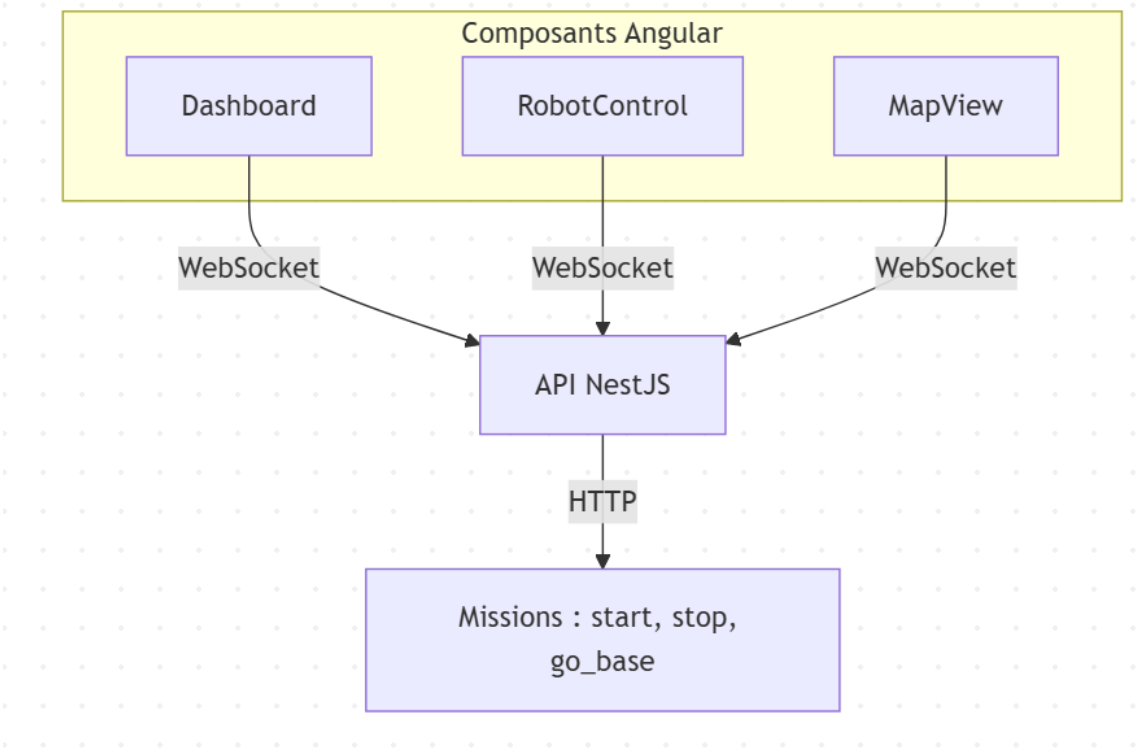


Figure 5 – Architecture de l'interface utilisateur Angular

DashboardComponent:

Supervision générale du système

Ce composant offre une vue d'ensemble en temps réel :

- État des robots (position, batterie, statut de mission)
- Nombre de robots connectés

- Logs système
- Résumé de la mission en cours

Les données sont mises à jour en temps réel via WebSocket, avec un rafraîchissement d'au moins 1 Hz (R.F.3).

RobotControlComponent:

Envoi de commandes aux robots

Ce composant permet à l'utilisateur de piloter les robots à distance en envoyant des commandes de mission spécifiques via l'interface web :

- **start_mission** : Démarrer une mission d'exploration.
- **end_mission** : Arrêter la mission en cours.
- **go_base** : Ordonner aux robots de retourner à la base.

Ces commandes sont transmises via l'**API NestJS** qui les relaie aux robots physiques ou simulés, selon le contexte d'exécution. L'usage des **WebSockets** permet une interaction quasi instantanée.

Ce composant est essentiel pour initier et superviser l'exécution des missions, dont l'un des aspects critiques est **l'affichage en continu de la position des robots sur la carte** pendant leur exécution — **conformément au requis R.F.9**.

MapViewComponent:

Visualisation des cartes et trajectoires

Affiche en temps réel :

- La carte 2D générée par les robots
- La position actuelle de chaque robot
- Le chemin parcouru (trace historique)
- Les obstacles détectés

Les cartes sont reçues via WebSocket depuis l'API et sont mises à jour dynamiquement dans le navigateur.

Ce composant permet une **visualisation fluide et claire** du déroulement d'une mission (R.F.8).

API NestJS – Serveur de communication

L'ensemble des composants communiquent avec l'API via :

- **WebSocket (Socket.IO)** pour les données temps réel (carte, état, logs)
- **HTTP** pour les commandes (start, stop, go_base)

L'API centralise les requêtes, applique les règles de validation, puis les redirige vers le backend ROS2 ou la base PostgreSQL.

3.6 Fonctionnement général (Q5.4)

Le fonctionnement du système se décompose en plusieurs **phases clés**, allant de l'initialisation des services à l'exécution des missions, en passant par les tests et la supervision. L'ensemble du système repose sur une architecture conteneurisée (Docker) pour la station, ROS2 pour les robots, et une interface web Angular accessible depuis n'importe quel poste local.

1. Initialisation du système

Récupération du projet

- Cloner le dépôt **GitLab** contenant le code source du frontend, du backend, des fichiers de simulation et du logiciel embarqué.

Lancement de la station au sol

- Exécuter `docker-compose up` dans le répertoire principal pour démarrer :
  - Le **serveur NestJS**
  - La **base de données PostgreSQL**
  - Le **serveur web** exposant l'interface Angular

2. Accès à l'interface utilisateur

- Ouvrir un navigateur web sur l'URL : `http://localhost:4200`
- L'interface Angular permet de :
  - Visualiser les cartes générées
  - Suivre les positions des robots
  - Démarrer / arrêter les missions
  - Contrôler le retour à la base
  - Consulter les journaux d'événements

3. Démarrage du logiciel embarqué

Sur chaque robot physique Limo :

- Lancer ROS2 avec les nœuds suivants :
  - `communication_node` (échange avec la station)

    - `navigation_node` (exécution de la mission)
    - `sensor_node` (traitement des capteurs)
- Le robot se connecte automatiquement à la station via ROS2 Topics.

4. Utilisation de la simulation (optionnelle)

En mode simulation, sans robots physiques :

- Lancer **Gazebo Fortress** avec un monde préconfiguré.
- Démarrer le **pont `ros_gz_bridge`** pour assurer la compatibilité ROS2.
- Lancer les nœuds ROS2 simulés (`Robot1_sim`, `Robot2_sim`).
- La station reçoit les données comme si elles venaient de robots réels.

5. Exécution d'une mission

- Depuis l'interface web :
    - Sélectionner ou créer une nouvelle mission
    - Lancer l'exploration (`start_mission`)
    - Suivre en temps réel la position et l'état de chaque robot
    - Arrêter la mission (`end_mission`) ou demander le retour à la base (`go_base`)
- Toutes les données (positions, cartes, logs) sont enregistrées automatiquement dans la base PostgreSQL.

6. Tests automatisés

Le projet intègre plusieurs niveaux de tests pour garantir sa robustesse :

- **Tests unitaires** :
    - Frontend : `npm test`
    - Backend : `npm run test`
    - ROS2 : `pytest`
- **Tests d'intégration** : Exécutés via ROS2 pour valider les échanges entre nœuds.
- **Tests de bout en bout (E2E)** : `npx cypress open` pour simuler des parcours utilisateur réels.

4. Processus de gestion

4.1 Estimations des coûts du projet (Q11.1)

Le coût total du projet est estimé en fonction de plusieurs facteurs :

- Le **temps de travail alloué** à chaque rôle au sein de l'équipe (jusqu'à 630 heures-personnes),
- Les **coûts matériels** requis pour le déploiement et la validation du système multi-robot,
- Et, conformément à l'appel d'offres (section 2.2), les **coûts de support et maintenance à moyen terme** après livraison du système.

Ressources humaines

Conformément aux exigences du projet, la charge de travail maximale autorisée est de **630 heures-personnes**. Avec une équipe composée de **six membres**, cela correspond à **105 heures de travail par personne**.

Notre équipe comprend :

- **1 coordinateur de projet**, chargé de la supervision, la planification, le suivi de projet, la documentation et les démonstrations.
- **5 développeurs-analystes**, responsables de l'architecture, du développement, des tests, de la simulation et de la validation.

Tableau 1 : Coût des ressources humaines

Rôle	Taux horaire ($CAN/h)	Temps alloué (h)	Coûts ($CAN)
Coordonnateur	145	105	15 225
Développeurs	130	525	68 250
Total estimé		**630**	**83 475**

Ressources matérielles

L'Agence Spatiale Canadienne (ASC) exige l'utilisation d'au moins deux robots **AgileX Limo Pro** pour le développement et la validation du système multi-robot. Le coût unitaire est estimé à 3 200 USD, soit 4 542,81 $CAN, sur la base d'un taux de change de 1,42 $CAN pour 1 USD en date du 13 février 2025.

Tableau 2 : Coûts matériels

Matériel	Coût unitaire ($CAN)	Quantité	Coût total ($CAN)
Robot AgileX Limo	4 542,81	2	9 085,62
Total estimé			**9 085,62**

Maintenance et support à moyen terme

L'appel d'offres (section 2.2) précise que la proposition doit inclure une **durée de support après livraison**, couvrant :

- L'**assistance à la reprise de système** en cas de panne logicielle
- La **maintenance préventive** (mises à jour, correctifs)
- Le **transfert de connaissances** à long terme (documentation technique, support aux utilisateurs)
- La **correction de bugs mineurs** et l'éventuelle adaptation à de nouvelles contraintes expérimentales

Une provision de **30 heures supplémentaires** est réservée à ce volet, à un taux moyen de **150 $CAN/h**, soit un total de **4 500 $CAN**.

Cette prévision assure une **continuité opérationnelle du système après sa livraison**, et répond directement aux attentes du client.

Le tableau suivant résume les coûts totaux du projet :

Tableau 3 : Coûts totaux du projet

Ressource	Coût ($CAN)
Ressources humaines	83 475
Matériel	9 085,62
Support et maintenance	4 500
Total estimé	**97 060,62**

Le coût total estimé pour la réalisation du projet est donc de 92 060,62 $CAN. Cette estimation inclut le temps de travail nécessaire pour la planification, le

développement, les tests, la validation et la documentation du système multi-robot, ainsi que les coûts matériels requis.
Elle intègre également une provision pour le support technique et la maintenance à moyen terme, couvrant les mises à jour logicielles, le transfert de connaissances et la correction d'éventuels bogues après livraison, conformément aux exigences de l'appel d'offres.

4.2 Planification des tâches (Q11.2)

La planification du projet repose sur une démarche **structurée, incrémentale et collaborative**, définie autour de **trois jalons clés** :

- **PDR (Preliminary Design Review)** – 14 février 2025 : Validation de la conception initiale
- **CDR (Critical Design Review)** – 28 mars 2025 : Vérification intermédiaire de la robustesse du système
- **RR (Readiness Review)** – 15 avril 2025 : Démonstration finale et validation du livrable complet

Chaque tâche du projet est alignée sur les **requis fonctionnels (R.F.)** et **contraignants (R.C. / R.Q. / R.M. / R.L.)** identifiés dans l'appel d'offres, assurant une traçabilité complète entre les activités planifiées et les objectifs du système.

Tableau 4 – Diagramme Gantt combiné avec jalons et allocation temporelle

Planification projet avec diagramme Gantt

Tâches principales	Janvier	Février	Mars	Avril (1-15)	Jalons
Analyse des exigences (R.M, R.L, R.Q)	■■■■■ ■■				PDR (14/02/2025)
Configuration GitLab & Docker (R.L.4)	■■■■■ ■■				PDR (14/02/2025)
Implémentation Identification robot (R.F.1)		■■■■■ ■■			PDR (14/02/2025)
Exécution des commandes mission (R.F.2)		■■■■■ ■■			PDR (14/02/2025)
Exploration autonome (R.F.4)		■■■■■ ■■			CDR (28/03/2025)
Évitement d'obstacles (R.F.5)			■■■■■ ■■		CDR (28/03/2025)
Communication inter-robots (R.F.19)			■■■■■ ■■		CDR (28/03/2025)
Stockage des missions (R.F.17)			■■■■■ ■■		CDR (28/03/2025)
Carte en temps réel (R.F.8)			■■■■■	■■■■■	RR (15/04/2025)
Mise à jour logicielle (R.F.14)				■■■■■ ■■	RR (15/04/2025)
Tests finaux et débogage (R.Q.2)				■■■■■ ■■	RR (15/04/2025)

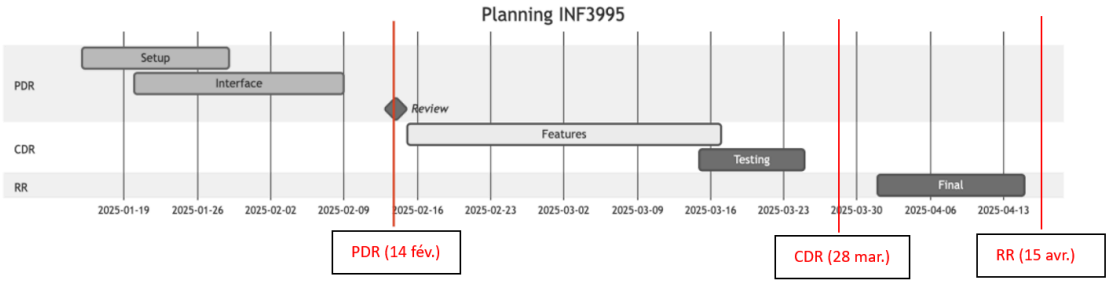
Le projet suit une logique d'**attribution par expertise**. Chaque membre de l'équipe est responsable d'un sous-ensemble de fonctionnalités ou de composants, ce qui garantit une **meilleure efficacité** et un **meilleur suivi qualité**.

Tableau 5 – Division des tâches par membre et par requis

Membre	Rôle	Responsabilités principales
Amine	Coordonnateur du projet	Gestion des ressources, suivi GitLab, validation des livrables
Kevin	Expert robotique & ROS	Implémentation des robots physiques *(R.F.1, R.F.2, R.F.4, R.F.5, R.F.19)*
Issam	Expert simulation & tests	Simulation Gazebo, tests d'exploration *(R.F.4, R.F.5, R.F.8)*
Yassine	Expert serveur & API	Gestion des services backend, communications *(R.F.2, R.F.19, R.F.14)*
Rafik	Expert base de données	Stockage et gestion des données *(R.F.17, R.F.18)*
Mario	Expert Frontend & UX	Interface utilisateur, affichage des cartes et données *(R.F.3, R.F.8, R.F.10, R.F.16)*

4.3 Calendrier de projet (Q11.2)

Tableau 6 – Planification globale par phase et jalon du projet INF3995



4.4 Ressources humaines du projet (Q11.2)

Notre équipe est composée de six membres aux compétences variées et complémentaires, assurant une couverture optimale de tous les aspects du projet. Elle inclut cinq programmeurs analystes et un coordonnateur de projet.

Les programmeurs analystes sont responsables de la conception, du développement, des tests et de la documentation du système. Ils assurent notamment l'intégration du frontend, l'optimisation des communications avec le serveur et la programmation des robots pour l'exécution des différentes missions.

Le coordonnateur de projet joue un rôle central dans la gestion des ressources, le suivi des tâches et le respect des échéances et des normes d'équipe. Plutôt qu'un modèle strictement centralisé, notre organisation repose sur un fonctionnement décentralisé favorisant la collaboration et l'autonomie.

Le suivi des tâches et des contributions est assuré via GitLab, garantissant une gestion efficace et transparente. Le tableau ci-dessous présente la répartition

des rôles, les qualifications et les principales responsabilités de chaque membre de l'équipe.

Tableau 7 – Répartition des rôles, compétences et responsabilités au sein de l'équipe

Rôle	Membre	Qualifications principales	Responsabilités principales
Coordonnateur de projet	Amine	Gestion de projet, suivi GitLab, revue de code	Supervision générale, gestion des ressources et des échéances, validation des contributions
Expert ROS et robotique	Kevin	Développement ROS, programmation robotique	Implémentation et optimisation des comportements des robots
Expert simulation et tests	Issam	Simulation Gazebo, tests en simulation	Développement et validation des tests en simulation
Expert serveur et tests	Yassine	Développement backend, tests unitaires	Implémentation et maintenance du serveur, validation des fonctionnalités
Expert serveur et base de données	Rafik	Bases de données, architecture serveur	Conception et gestion des bases de données, intégration backend
Expert frontend et UX	Mario	Développement web, UX/UI, communication serveur	Conception et optimisation de l'interface utilisateur, intégration frontend-backend

Il est à noter que tous les membres participent au développement sous ROS, et les responsabilités peuvent évoluer selon les besoins du projet.

5. Suivi de projet et contrôle

5.1 Contrôle de la qualité (Q4)

La qualité des biens livrables est essentielle pour garantir la fiabilité et la robustesse du projet.

Notre équipe met en place un processus rigoureux de révision et de validation à plusieurs niveaux afin d'assurer la conformité des livrables aux exigences du

projet, tout en respectant des **bonnes pratiques de développement** et un **guide de style commun** à l'équipe.

1. Révision des livrables clés : PDR, CDR et RR

Chaque phase du projet (Preliminary Design Review, Critical Design Review et Readiness Review) est soumise à une révision approfondie selon les critères suivants :

- **Vérification de la complétude** : Chaque livrable est évalué en fonction des requis attendus (fonctionnels, matériels, logiciels, qualité, etc.).
- **Relecture par l'équipe** : Tous les membres du groupe doivent passer en revue les documents et apporter des corrections avant la soumission.
- **Feedback et corrections** : Des itérations sont effectuées en fonction des retours reçus, afin d'améliorer la clarté et la précision du contenu.

2. Utilisation des Merge Requests et d'une norme de commits

Afin d'assurer un code de qualité et une gestion efficace du développement logiciel, nous mettons en place les pratiques suivantes :

- **Merge Requests (MR)** : Toute modification du code doit passer par une MR, qui sera revue par au moins un autre membre de l'équipe avant d'être fusionnée dans la branche principale.
- **Norme de commits** : Les commits doivent suivre un format standardisé (ex. *feat: ajout de la détection d'obstacles* ou *fix: correction d'un bug d'affichage*), permettant une meilleure traçabilité et compréhension de l'évolution du projet.
- **Revue de code systématique** : Chaque MR doit être accompagnée d'une description claire des changements et de leur impact, et être validée par au moins un autre membre.
- **Application d'un guide de style commun** : L'équipe applique un guide de style commun couvrant l'ensemble des langages utilisés (TypeScript pour Angular/NestJS, Python pour ROS2). Ce guide définit des règles strictes de nommage (camelCase, PascalCase, snake_case), une structure de projet cohérente par fonctionnalité, et l'usage systématique d'outils automatiques tels que Prettier [3] pour le Angular, Black [4] et pylint [5] pour le Python. Il impose également une **convention de commits standardisée** inspirée de **Conventional Commits** [6] pour une meilleure traçabilité. Ces règles assurent une cohérence du code, facilitent le travail collaboratif et garantissent un niveau élevé de qualité logicielle.
- **Bonnes pratiques de développement** : Modularité du code, séparation des responsabilités, documentation interne et gestion des erreurs font partie intégrante des standards appliqués.

3. Tests et validation continue

- **Tests unitaires** : Chaque composant logiciel doit être testé individuellement pour assurer son bon fonctionnement.
- **Tests d'intégration** : Des tests sont effectués régulièrement pour s'assurer de la compatibilité entre les différentes composantes du système (interface utilisateur, robots, simulation).
- **Démonstrations vidéo** : Chaque fonctionnalité clé est validée par une démonstration filmée, permettant de vérifier son bon fonctionnement et de fournir une preuve de conformité aux exigences.

Grâce à ces pratiques, nous nous assurons que les livrables sont de qualité et conformes aux attentes du projet.

5.2 Gestion de risque (Q11.3)

Tout projet comporte son lot de risques. L'important est de **les identifier dès le départ**, d'évaluer leur probabilité et leur impact, et de mettre en place des **stratégies d'atténuation adaptées**.
Dans ce projet, plusieurs risques techniques, organisationnels et humains ont été identifiés.

Le tableau ci-dessous présente une **synthèse des risques les plus pertinents**, accompagnée de leur **probabilité d'occurrence**, de leur **criticité** (impact potentiel), ainsi que des **mesures mises en place pour les atténuer**.

Tableau 8 – Synthèse des risques identifiés avec probabilité, criticité et stratégies d'atténuation

Risque	Probabilité	Criticité	Stratégies d'atténuation
Précision insuffisante des robots	Moyenne	Élevée	Tests physiques répétés, calibration des algorithmes, ajustements du code embarqué
Limitation du réseau WiFi	Élevée	Moyenne	Optimisation des données, reconnexion automatique, tests de bande passante
Respect des délais	Moyenne	Élevée	Jalons intermédiaires, marge pour imprévus, gestion agile avec revues régulières
Intégration système difficile	Faible	Élevée	Interfaces claires définies dès le départ, tests d'intégration fréquents
Bris matériel	Faible	Moyenne	Manipulation prudente, recours à la simulation Gazebo en cas de panne
Indisponibilité d'un membre	Faible	Élevée	Répartition des tâches, documentation détaillée, absence de "points uniques de défaillance"
Conflits internes	Faible	Moyenne	Communication ouverte, décisions collectives, médiation si nécessaire
Difficultés techniques imprévues	Moyenne	Élevée	Priorisation des tâches complexes, solutions alternatives prévues, consultation d'experts

Ces stratégies permettent à notre équipe de **réduire les incertitudes**, de **réagir efficacement aux imprévus** et d'assurer un bon déroulement du projet jusqu'à la livraison finale.

5.3 Tests (Q4.4)

Les tests effectués dans ce projet visent à garantir la fiabilité, la performance et l'interopérabilité entre les différentes composantes du système. Ils couvrent à la fois le **matériel** (robots) et le **logiciel** (client, serveur, simulation).

1. Tests du Client (Frontend – Angular)

Objectif : Vérifier l'affichage, l'interaction et la communication avec le backend.

- **Tests unitaires** : Vérification des composants et services (*.spec.ts avec Jasmine/Karma).
    - Ex. : Test du composant de contrôle des robots (control-panel.component.spec.ts).
- **Tests d'interface utilisateur** : Validation de l'affichage et de la navigation dans l'interface.

- **Tests d'intégration** : Simulation des requêtes entre le frontend et le backend.

2. Tests du Serveur (Backend – NestJS)

Objectif : Vérifier le bon fonctionnement des API REST et WebSockets.

- **Tests unitaires** : Validation des services (*.service.spec.ts avec Jest).
  - Ex. Vérification du robot.service.ts pour l'envoi des commandes aux robots.
- **Tests API** : Vérification des endpoints avec Postman ou Supertest (*.controller.spec.ts).
- **Tests de performance** : Mesure des temps de réponse sous charge.

3. Tests des Robots (Système embarqué – ROS2)

Objectif : Assurer le bon fonctionnement des robots physiques.

- **Tests de navigation** : Vérification du respect des trajectoires définies.
- **Tests de détection d'obstacles** : Validation de l'évitement en fonction des capteurs.
- **Tests de communication** : Vérification de l'échange de messages entre les robots et la station au sol.

4. Tests de la Simulation (Gazebo)

Objectif : Vérifier la cohérence entre la simulation et le monde réel.

- **Tests d'exploration** : Vérification de l'algorithme de déplacement autonome.
- **Tests de cartographie** : Comparaison des cartes générées avec l'environnement réel.

5.4 Gestion de configuration (Q4)

Système de Contrôle de Version

L'équipe utilise **GitLab** avec une gestion rigoureuse du code source et de la documentation.

- **Branches principales :**
  - master : Code stable, protégé contre les modifications directes.
  - develop : Intégration continue des nouvelles fonctionnalités avec tests automatisés.
- **Branches spécifiques :**

    - feature/ : Nouvelles fonctionnalités (ex. feature/robot-identification).
    - hotfix/ : Corrections urgentes fusionnées vers master et develop.
- **Commits :** Messages clairs en français, commits atomiques, référence aux issues GitLab.

Organisation du Code Source

Le projet suit une architecture modulaire :

- **Client (/client)** : Frontend Angular (composants, services, modèles).
- **Serveur (/server)** : Backend NestJS (API REST, WebSockets, logique métier).
- **Robot (/robot)** : Système ROS2 (contrôle, communication, navigation).

Organisation des Tests

- **Unitaires** : Tests pour chaque composant (Jasmine/Karma pour Angular, Jest pour NestJS).
- **Intégration** : Tests API et WebSockets simulant des scénarios réels.
- **Robotiques** : Vérification des capteurs et de la navigation autonome.
- **CI/CD** : Tests automatisés et déploiement continu via GitLab CI/CD.

Gestion des Données

- **Base de données PostgreSQL** : Enregistrement des missions, positions des robots et métriques de performance.
- **Fichiers de configuration** : Paramètres des robots et de la simulation centralisés en YAML.
- **Logs et métriques** : Historique détaillé des missions et suivi des performances.

Documentation

- **Techniques** : ARCHITECTURE.md, diagrammes, spécifications détaillées.
- **API** : Documentation intégrée aux fichiers source.
- **Installation** : Instructions de build et déploiement (README.md).
- **Utilisation** : Guides pour interface utilisateur et dépannage.

Cette approche assure une séparation claire des responsabilités et une traçabilité efficace du projet.

5.5 Déroulement du projet (Q2.5)

Globalement, l'équipe a su maintenir une bonne dynamique de travail, mais certains ajustements ont été nécessaires au fil de l'avancement.

Ce qui a bien fonctionné :

- L'analyse des requis (R.M, R.L, R.Q) en début de session a été rigoureuse, permettant de poser des bases solides.
- La séparation claire entre les modules backend, frontend, simulation et embarqué a facilité la répartition du travail.
- L'interface utilisateur Angular a été développée rapidement et a permis des tests de visualisation très tôt dans le projet (R.F.3, R.F.10).
- Les missions de base (start, stop, go_base – R.F.2) ont été testées avec succès sur les robots simulés, puis sur le matériel réel.
- Le système de communication en WebSocket entre l'interface et le backend NestJS s'est révélé efficace pour gérer les données en temps réel (R.C.1).
- L'équipe a suivi une organisation agile, avec des revues hebdomadaires et un suivi clair via GitLab (issues, merge requests, milestones).

Ce qui a été plus difficile ou imprévu :

- L'accès au laboratoire pour manipuler les robots AgileX Limo a été plus restreint que prévu, ce qui a retardé les premiers tests physiques, notamment car ce n'est pas tous les robots qui étaient fonctionnels.
- La calibration des capteurs (Lidar, caméra, IMU) a nécessité plusieurs essais, car les comportements observés différaient entre simulation et réalité.
- La communication réseau entre la station au sol et les robots s'est révélée instable à certains moments. Des pertes de messages ou des délais ont été observés lors des tests multi-robot, notamment dans les cas où les deux robots étaient en mouvement simultané (R.F.19).
- La séparation de la logique entre les deux robots pour avoir deux canaux de communication indépendants a été difficile en raison de l'interférence de ROS 2 (tous les messages sont envoyés sur le même réseau).
- Certains algorithmes comme l'exploration autonome (R.F.4) ou l'évitement d'obstacles (R.F.5) ont demandé plus de travail que prévu, et leur fiabilité reste variable selon les environnements testés.
- Nous n'avons pas pu filmer nos explorations avec 2 limos à la fois. Cependant, le système est bel et bien fonctionnel (voir annexe)

Malgré ces défis, l'équipe a su s'adapter rapidement. Des alternatives ont été mises en place lorsque nécessaire, ce qui a permis de respecter tous les jalons du projet. Des ajustements ont été réalisés en cours de route afin de maintenir un bon rythme tout en assurant la traçabilité des limites du système.

6. Résultats des tests de fonctionnement du système complet (Q2.4)

Les tests réalisés sur la version intégrée du système montrent que la majorité des fonctionnalités clés sont opérationnelles et répondent aux requis spécifiés dans l'appel d'offres.

Fonctionnalités validées :

- **Gestion des missions** : Les actions *start*, *stop* et *go_base* sont correctement interprétées par les robots et exécutées sur le terrain comme en simulation (R.F.2).
- **Interface Web** : L'application Angular permet d'envoyer les commandes, visualiser l'état du robot, consulter les journaux d'activité, et afficher la carte en temps réel (R.F.3, R.F.8, R.F.10).
- **Communication WebSocket** : L'échange de données entre la station au sol (NestJS) et les robots via ROS2 Topics est fonctionnel, y compris pour les informations de position, capteurs, et statuts de mission (R.F.9, R.C.1).
- **Exploration et obstacles** : Les robots détectent correctement certains obstacles simples et peuvent éviter les collisions dans des environnements peu encombrés (R.F.4, R.F.5).
- Le système est conçu pour fonctionner avec un ou deux robots. La station au sol et l'interface utilisateur s'adaptent automatiquement en se connectant à tous les robots disponibles pour la mission, sans que l'utilisateur ait besoin d'en préciser le nombre (R.C.5).

Fonctionnalités partiellement fonctionnelles ou non finalisées :

- La visualisation avancée de la carte (R.F.8) n'est pas complètement intégrée à l'interface : les données sont disponibles mais la représentation graphique est limitée.
- Le mode de contrôle sur les roues Ackerman n'a pas encore été implémenté (R.F.15).
- La base de données présente sur la station au sol est partiellement implémentée. En effet, les informations des anciennes missions ne sont pas encore affichées, le tri des missions n'est donc pas encore possible (R.F.17).
- La génération de la carte lors d'une mission sur la station au sol n'a pas encore été implémentée (R.F.18).
- Chaque robot ne communique pas encore en continu aux robots sa distance à son point de départ ou à la station au sol (au choix du contractant) (R.F.19).
- Le comportement autonome en environnement inconnu reste perfectible : le robot peut se bloquer dans certains cas limites, notamment en présence d'obstacles très rapprochés.
- La stabilité réseau affecte parfois la réactivité de la carte ou des journaux affichés en temps réel.
- **Multi-robot** : La communication P2P entre deux robots n'est pas encore possible, et des missions indépendantes ne peuvent pas encore être assignées à chacun (R.F.19).

Dans l'ensemble, les **requis essentiels pour la démonstration** sont atteints. Les écarts restants sont connus, documentés, et des solutions d'amélioration sont envisagées pour la suite.

7. Références (Q3.2)

[1] Randstad, *Qu'est-ce que la méthode agile, et pourquoi est-elle si répandue dans le monde des TI ?*, 26 juillet 2023. [En ligne]. Disponible sur : https://www.randstad.ca/fr/employeurs/tendances-employeur/innovation-en-milieu-de-travail/quest-ce-que-la-methode-agile-et-pourquoi-est-elle-si-repandue-dans-le-monde-des-ti/

[2] Agilex. "limo-pro." https://global.agilex.ai/products/limo-pro (accédé le 13 février 2025).

[3] Prettier, *What is Prettier?*, [en ligne]. Disponible : https://prettier.io/docs/ (consulté le 10 février 2025).

[4] Black, *The uncompromising Python code formatter*, [en ligne]. Disponible : https://black.readthedocs.io/en/stable/ (consulté le 10 février 2025).

[5] pylint, *Pylint 3.3.6 documentation*, [en ligne]. Disponible : https://pylint.pycqa.org (consulté le 10 février 2025).

[6] ConventionalCommits.org, Commits Conventionnels [en ligne]. Disponible : https://www.conventionalcommits.org/fr/ (consulté le 10 février 2025).

ANNEXES

Raison de la démo avec 1 robot physique.

Malheureusement, malgré que notre système d'exploration fonctionne avec 2 robots (2 robots gazebo, 2 limos AGILE), les jours avant la remise la salle était trop remplie et trop peu de robots avec les lidars étaient disponibles. Notre implémentation est fonctionnelle avec 2 robots gazebo ou 2 limos (même code utilisé pour les deux : robot/robot/src/mission/mission/mission.py). Voici comment lancer 2 limos AGILE en exploration pour tester notre système.

-Lancer le script *'robot1.sh'* ou *'robot2.sh'* situés dans *robot/robot_launch_scripts*, qui lancera le système d'exploitation et de communication avec la base selon les canaux limo1/x ou limo2/x. Ce script sépare donc la logique de communication et de la mission en 2 robots séparés.

-Lancer le script *'start-all-1.sh'* ou *'start-all-2.sh'* situés dans le répertoire *robot/limo_launch_scripts*, qui lancera la limo dans son namespace spécifique. ce script lancera donc limo1/x et limo2/x, pour séparer le cartographe, lidar, odom, cmd_vel, limo_status, nav2, …

-Pour l'instant, beaucoups de fichiers sont dédoublés pour simplifier la logique de contrôle des 2 robots, mais on prévoit supprimer la duplication de code dans le future (comme par exemple, nav2_config1.yaml et nav2_config2.yaml)

-Ensuite il ne reste plus qu'à lancer la station au sol avec le script fait à cet effet, puis l'interface utilisateur avec 'npm start'.

Regroupement des Requis

1. Requis Généraux

- **R.G.1** : Le système doit permettre l'exploration autonome d'une pièce par une équipe de robots.
- **R.G.2** : Une station au sol doit permettre la supervision et l'interaction avec les robots.

2. Requis Matériels

- **R.M.1** : Utilisation obligatoire des robots AgileX Limo.
- **R.M.2** : Communication unique via le réseau WiFi fourni par l'Agence.
- **R.M.3** : Seuls les capteurs fournis par l'Agence sont autorisés.
- **R.M.4** : La station au sol doit être un laptop ou un PC.

3. Requis Logiciels

- **R.L.1** : OS Ubuntu requis pour les robots, machine virtuelle autorisée.
- **R.L.2** : Interface utilisateur identique pour simulation et robots physiques.
- **R.L.3** : Commandes de haut niveau seulement, pas de contrôle direct des robots.
- **R.L.4** : Conteneurisation Docker obligatoire pour toutes les composantes logicielles, sauf pour les robots physiques.

4. Requis Fonctionnels

- **R.F.1** : Identification individuelle des robots via l'interface utilisateur.
- **R.F.2** : Commandes "Lancer la mission" et "Terminer la mission" disponibles.
- **R.F.3** : Affichage de l'état des robots à une fréquence d'au moins 1 Hz.
- **R.F.4** : Exploration autonome de l'environnement.
- **R.F.5** : Évitement des obstacles détectés.
- **R.F.6** : Retour à la base avec précision de 0,3 m.
- **R.F.7** : Retour automatique à la base en cas de batterie faible (<30%).
- **R.F.8** : Génération d'une carte de l'environnement en temps réel.
- **R.F.9** : Affichage continu de la position des robots.
- **R.F.10** : Interface utilisateur accessible sur PC, tablette et téléphone.
- **R.F.11** : Carte en 3D et en couleur.
- **R.F.12** : Spécification de la position initiale des robots.
- **R.F.13** : Détection et évitement des élévations négatives.
- **R.F.14** : Mise à jour du logiciel de contrôle des robots via l'interface utilisateur.
- **R.F.15** : Support de deux modes de contrôle des roues.
- **R.F.16** : Éditeur de code intégré pour modifier le comportement des robots.
- **R.F.17** : Base de données enregistrant les missions.
- **R.F.18** : Sauvegarde et inspection des cartes générées.
- **R.F.19** : Communication P2P entre robots pour transmettre la distance à la base.
- **R.F.20** : Implémentation d'une zone de sécurité (geo-fence).

5. Requis de Conception

- **R.C.1** : Disponibilité des logs de débogage en continu.
- **R.C.2** : Déploiement du logiciel avec une seule commande.
- **R.C.3** : Génération aléatoire de l'environnement virtuel dans Gazebo.
- **R.C.4** : Interface utilisateur respectant les heuristiques de Nielsen.
- **R.C.5** : Compatibilité avec un ou deux robots sans configuration manuelle.

6. Requis de Qualité

- **R.Q.1** : Code standardisé suivant des conventions reconnues.
- **R.Q.2** : Tests unitaires ou procédures de test pour chaque fonctionnalité.