ChatGPT

Plan d'implémentation du back-end TravauxHub sur Lovable.dev

1. Vue d'ensemble du projet

Le back-end de TravauxHub vise à mettre en place une plateforme CRM pour la gestion des artisans et de leurs chantiers. L'objectif est de **tirer parti de Lovable.dev**, une solution no-code/low-code basée sur Supabase (PostgreSQL) et des Edge Functions, afin de réaliser rapidement un back-end robuste. Ce plan détaille la structure de données nécessaire, les fonctionnalités clés (pipeline de leads, tableau de bord artisans, scoring, suivi de chantier, notifications e-mail), ainsi que la logique d'automatisation possible avec Lovable.dev. L'approche privilégiera les solutions **sans code** lorsque c'est possible (en exploitant les workflows natifs de Lovable) et identifie les éléments requérant des fonctions avancées ou des interventions manuelles.

2. Structure de la base de données

Pour supporter les fonctionnalités du CRM TravauxHub, nous prévoyons les tables principales suivantes : **Artisans**, **Leads**, **Assignations**, **Chantiers**, **Documents**, **Devis**, **Messages** et **Événements**. Ces tables et leurs relations seront créées via Lovable.dev (qui utilise Supabase en backend). Lovable permet de décrire simplement les données pour générer les tables sans écrire de SQL [1]. Ci-dessous, chaque entité et ses champs essentiels :

- **Artisans** : Profil des artisans inscrits. Champs clés : identifiant (UUID), nom, email, téléphone, mot de passe (géré par l'authentification Lovable/Supabase), métier (activité professionnelle), zone d'intervention (base géographique et rayon en km), indicateur RGE (oui/non ou type de certification), assurance (oui/non, date expiration), date d'inscription, statut (actif/en attente). On peut stocker également des métriques de performance comme le taux de conversion ou la réactivité (voir section scoring). Chaque artisan correspond à un compte utilisateur dans l'auth système de Lovable [2].

- **Leads** : Représente des demandes de travaux entrantes (prospects). Champs : identifiant, date de création, nom et coordonnées du client/prospect (téléphone, email), localisation (adresse/ville, éventuellement lat/long), description du projet/travaux, catégorie de travaux (métier requis), **score du lead** (note calculée selon fraîcheur et complétude, voir section scoring), statut global. *Remarque :* le statut global du lead peut simplement indiquer s'il est « ouvert » ou « fermé », car le détail d'avancement est géré par assignation (un lead peut être traité par plusieurs artisans). Un lead est lié à une ou plusieurs assignations vers des artisans.

- **Assignations** : Table pivot reliant un **Lead** à un **Artisan**, créé lorsqu'un lead est distribué à un artisan. C'est au niveau de l'assignation qu'on suit le **pipeline** du lead pour cet artisan. Champs : identifiant, **lead_id** (référence au lead concerné), **artisan_id** (référence de l'artisan destinataire), statut (étape actuelle dans le pipeline pour cet artisan : « Nouveau », « En contact », « Devis », « Gagné » ou « Perdu »), dates clés (date d'assignation, date de premier contact, date d'envoi du devis, date de conclusion gagnée/perdue), éventuellement un score contextuel (par ex. score du lead au moment de l'assignation). Cette table permet à chaque artisan de suivre ses leads de

manière indépendante. **Relations** : un lead peut avoir plusieurs assignations (s'il est envoyé à plusieurs artisans), et chaque assignation concerne un artisan unique.

- **Chantiers** : Représente un projet confirmé (démarrage de travaux) issu d'un lead **gagné** par un artisan. Quand une assignation passe à l'étape « Gagné », on crée un enregistrement de chantier pour suivre l'exécution. Champs : identifiant, référence à l'assignation gagnée (ou lead_id + artisan_id), statut du chantier (ex : « Préparation », « En cours », « Terminé »), date de début prévue/réelle, date de fin prévue/réelle, lien vers le **devis signé** (par ex. champ booléen ou référence document confirmant que le devis est accepté), notes de suivi. **Relations** : un chantier est associé à un couple lead+artisan unique (issue d'une assignation gagnée). On pourra lier les photos et docs de chantier via la table Documents.

- **Documents** : Stocke les fichiers et justificatifs uploadés sur la plateforme (utilise la **Storage** de Lovable/Supabase pour héberger les fichiers [3]). Champs : identifiant, fichier (URL ou chemin dans le bucket de stockage), type de document (par ex. « Certificat RGE », « Attestation assurance », « Photo chantier », « Devis PDF », etc.), date d'upload, date d'expiration (si applicable, pour documents RGE/assurance), statut de validation (ex : approuvé ou en attente de vérification). **Relations** : peut référencer soit un artisan (documents administratifs de l'artisan) soit un chantier (documents liés au projet). On peut prévoir un champ optionnel artisan_id et/ou chantier_id selon le type, ou bien deux tables séparées si préféré (mais un champ « owner_type » / « owner_id » polymorphe est flexible). La vérification de certains documents (RGE, assurances) sera faite manuellement par un admin, avec mise à jour du statut de validation.

- **Devis** : Table des devis émis par les artisans pour les leads. Champs : identifiant, reference de l'assignation (lead+artisan) concernée, montant du devis, détails/description, date d'émission, statut (envoyé, accepté, refusé). Un champ fichier peut contenir un PDF du devis si l'artisan en upload un, sinon on stocke les détails textuels. **Relations** : un devis appartient à une assignation. (Option simplifiée : on peut permettre un seul devis par assignation, ou autoriser plusieurs versions – dans ce cas cette table aura potentiellement plusieurs entrées par assignation). Quand un devis est accepté par le client, on marque le devis « accepté » et l'assignation correspondante passe à « Gagné » (ce qui crée le chantier).

- **Messages** : Historique des messages ou communications. Ceci peut inclure des messages envoyés via la plateforme (par ex. si on implémente une messagerie entre artisan et client), ou simplement des **notes internes** et e-mails automatiques. Champs : identifiant, type (par ex. « message artisan-client » ou « notification système »), contenu du message, expéditeur (artisan, client, système), destinataire, date/heure. **Relations** : on peut lier un message soit à un lead/ assignation (conversation liée à un prospect) soit à un chantier (communications en phase travaux) selon le contexte. *NB:* Si la plateforme ne propose pas de chat intégré, cette table peut au moins journaliser les emails envoyés (copies des notifications) ou servir de champ "notes" pour les artisans.

- **Événements** : Journal des activités et changements d'état, pour audit et affichage d'un timeline. Champs : identifiant, type d'événement (ex: « Lead créé », « Lead assigné à X », « Statut changé: En contact », « Devis envoyé », « Chantier marqué terminé »…), description ou métadonnées (par ex. ancien et nouveau statut), auteur (utilisateur ou système à l'origine), timestamp. **Relations** : un événement peut référencer une entité concernée (artisan_id, lead_id, assignation_id, ou chantier_id selon le cas). Cette table permet d'afficher une frise chronologique dans le dashboard (ex: "Le lead #123 est passé à l'étape Devis le 05/11/2025 par Artisan X").

En résumé, ces tables couvrent l'ensemble des données nécessaires. Les clés étrangères et index seront utilisées pour relier artisans↔leads (via assignations), assignations↔devis, assignations↔chantiers, etc. Avec Supabase (PostgreSQL), on pourra définir ces relations et les exploiter dans les requêtes ou les sécuriser par RLS si besoin. Lovable générera ces tables sur la base de notre description, ce qui évite les manipulations SQL manuelles [4].

3. Gestion des artisans : inscription, profil et dashboard KPI

Inscription et authentification : On s'appuiera sur l'authentification native de Lovable/Supabase pour gérer les comptes artisans de façon sécurisée [2]. Lovable peut générer les pages de signup/login sans code (email/password, voire Google OAuth) [5]. Lors de l'inscription, on collectera des informations via un formulaire initial (après la création du compte utilisateur) pour compléter le profil artisan. Ce formulaire d'onboarding comprendra : - **Métier/Spécialité :** sélection du type de travaux réalisés (e.g. plombier, électricien, maçon...). - **Zone d'intervention :** choix d'une ville principale et d'un rayon d'action (en km). On pourra utiliser un champ texte pour la ville (éventuellement auto-complété) et un champ numérique pour le rayon. Ces données serviront à la géolocalisation et à la distribution des leads. - **Documents à fournir :** upload de justificatifs comme l'attestation d'assurance décennale et, si applicable, la **certification RGE**. Chaque fichier sera stocké dans la table Documents avec type approprié (et champ d'expiration pour suivre leur validité). Lovable facilite l'upload et la gestion de fichiers via son stockage intégré, sans code nécessaire [3]. Un statut « en attente de validation » sera associé à ces documents jusqu'à vérification manuelle par un administrateur.

Une fois inscrit, l'artisan pourra se connecter à son **tableau de bord**. Ce dashboard, accessible uniquement après login (on protégera la route via Lovable/Supabase Auth [6]), présentera plusieurs éléments clés : - **Liste de leads reçus** : affichage des leads (via les assignations) qui ont été distribués à cet artisan, avec leur statut (Nouveau, En contact, etc.), la date de réception, le nom/localisation du prospect, et éventuellement le score du lead. L'artisan pourra cliquer un lead pour voir les détails et les actions possibles (voir section pipeline). - **KPI de performance** : indicateurs chiffrés pour aider l'artisan à suivre son efficacité. Par exemple : nombre de leads reçus ce mois, nombre de devis envoyés, taux de conversion (pourcentage de leads gagnés), et un indicateur de **réactivité** (p. ex. temps moyen entre réception d'un lead et premier contact). Ces métriques peuvent être calculées à partir des données d'assignations et de devis, puis affichées sous forme de tuiles ou de graphiques simples. Lovable permet d'intégrer des composants visuels (ex: graphiques via Highcharts ou D3.js [7]) si besoin pour les KPI, mais on peut commencer par des chiffres basiques. - **Suivi des documents** : sur le dashboard artisan, indiquer l'état de ses documents administratifs (validés/expirant). Par exemple, un avertissement si son attestation d'assurance va expirer dans le mois, ou si la certification RGE n'est pas à jour. On peut calculer le temps restant et afficher un message incitatif à mettre à jour. *(L'envoi d'e-mails automatiques pour rappel de documents expirés est détaillé plus loin.)*

Le **profil artisan** pourra être édité sur une page dédiée où il peut modifier ses informations (adresse, rayon, etc.) et ajouter/mettre à jour ses documents. Grâce à Lovable, gérer ces formulaires de profil est relativement simple (CRUD auto-généré). Il faudra s'assurer de contrôler les formats (ex: numéro de téléphone valide, champs obligatoires remplis), ce que Lovable peut faire via des validations front-end sur mesure [8].

Enfin, un administrateur de TravauxHub aura possiblement une vue back-office des artisans (liste, filtres par métier/zone, vérification des docs). Ce point n'était pas explicitement demandé, mais à prévoir pour la modération. On pourrait gérer ça via une interface admin (protégée et réservée) ou directement en base si l'équipe choisit de faire la vérification hors application. Quoi qu'il en soit, la table Artisans comporte des flags (docs validés, etc.) modifiables par un admin.

4. Pipeline des leads et assignations

Le cœur du CRM est le **pipeline de gestion des leads**, qui suit chaque opportunité depuis son arrivée jusqu'à son aboutissement (conversion ou perte). TravauxHub définit les étapes suivantes : **Nouveau → En contact → Devis → Gagné / Perdu**. Ce sont des étapes classiques d'un cycle de vente d'affaire [9]. Chaque lead reçu par un artisan (représenté via une assignation) évoluera à travers ces statuts.

Création et assignation des leads : Les leads pourront provenir de différentes sources (formulaire en ligne TravauxHub rempli par un particulier, démarchage, import, etc.). Lorsqu'un nouveau lead est créé dans la table **Leads**, le système doit le **router vers des artisans appropriés**. Le critère principal de routage sera le métier requis et la zone géographique : - On identifie les artisans dont le métier correspond à la demande du lead, *et* dont la zone couvre l'adresse du client (par exemple, en calculant la distance entre la ville du lead et la ville de l'artisan, ou en comparant les codes postaux/région). Si la distance est inférieure au rayon d'intervention de l'artisan, on considère qu'il peut prendre le lead. - Pour chacun de ces artisans éligibles, on crée une entrée dans **Assignations** liant l'artisan et le lead, avec statut initial « Nouveau » et date d'assignation courante. Ainsi, chaque artisan reçoit effectivement le lead dans son pipeline personnel.

Remarque : on peut décider de limiter le nombre d'artisans assignés par lead (par exemple, envoyer chaque lead aux 3 meilleurs artisans correspondants, pour éviter une trop grande concurrence). Ce critère « top artisans » pourrait se baser sur le **scoring artisan** (voir section suivante), par exemple ne choisir que ceux ayant un bon taux de signature. Ce raffinement n'est pas explicitement demandé, mais peut être envisagé dans le code d'assignation si besoin.

Suivi des étapes (Nouveau, En contact, Devis, etc.) : Une fois assigné, chaque artisan doit pouvoir mettre à jour l'état d'un lead selon sa progression: - **Nouveau** : lead non encore traité (par défaut à l'assignation). Idéalement l'artisan contacte rapidement le prospect. - **En contact** : l'artisan a réussi à échanger avec le prospect (appel, visite, etc.). Cette étape peut être déclenchée manuellement par l'artisan via l'interface (en cliquant « Marquer comme contacté ») une fois le contact établi. - **Devis** : l'artisan a réalisé un devis pour le prospect. Ici, on peut automatiser le passage à l'étape Devis dès qu'un devis est généré ou envoyé dans le système. Par exemple, **lorsque l'artisan télécharge un devis** ou remplit un formulaire de devis et clique "Envoyer", on peut automatiquement mettre à jour le statut en « Devis » [10]. Cela évite une action manuelle redondante et assure que le pipeline reflète la réalité (cette auto-mise-à-jour sera implémentée via un workflow ou trigger côté back-end). - **Gagné** : le prospect a accepté le devis de l'artisan, le chantier est remporté. L'artisan devrait marquer le lead comme gagné dès qu'il a confirmation du client (ou bien l'admin peut le faire). On peut proposer un bouton « Marquer comme gagné » qui déclenche alors la création d'un enregistrement dans **Chantiers** pour démarrer le suivi de chantier. *Conséquence :* si un lead est gagné par un artisan, les autres artisans assignés au même lead devraient être marqués « Perdu » (soit automatiquement, soit manuellement par admin en clôturant les autres offres). On veillera à mettre à jour toutes les assignations du lead en conséquence de la victoire de l'un d'eux, probablement via une fonction serveur. - **Perdu** : le chantier n'a pas été remporté par cet artisan (soit le client a choisi un concurrent, soit n'a pas donné suite). L'artisan peut marquer « Perdu » s'il apprend qu'il n'a pas eu le marché, ou bien l'admin pourra clôturer les leads restés sans suite après un certain temps.

Le pipeline pourra être visualisé en **vue Kanban** (colonnes par étape) dans l'interface artisan, ou simplement via des filtres/badges à côté des leads. Une vue Kanban est très parlante pour suivre l'avancement [11], mais cela dépend des composants UI disponibles. Lovable.dev devrait permettre de créer une telle vue (via un composant custom ou en listant par statut). Dans tous les cas, **les transitions d'étapes peuvent être gérées** soit par une action utilisateur (drag & drop dans un Kanban, bouton de

changement) soit automatiquement suite à certains événements (comme vu pour l'étape Devis ou la conclusion) [10] . On implémentera des règles métier pour ces automatisations : - *Exemple:* Si un devis est créé/émis pour l'assignation X, alors statut X → « Devis » automatiquement [10] . - *Exemple:* Si un devis de X est marqué "accepté", alors X → « Gagné » et toutes les autres assignations du même lead → « Perdu ».

Ces règles pourront être réalisées via des **Edge Functions** (par exemple, une fonction Supabase déclenchée à l'insertion d'un devis, qui met à jour le statut) ou via des triggers SQL sur la base. Lovable permet de créer facilement des Edge Functions en JavaScript/TypeScript à déployer sur Supabase [12] [13] .

Scoring automatique des leads et artisans : Deux types de scoring sont mentionnés : le score du lead (qualité de l'opportunité) et le score de l'artisan (performance du professionnel). Ces scores seront stockés et mis à jour automatiquement par le système pour orienter les priorités.

- **Score du lead** : calculé à la création du lead (et éventuellement mis à jour par la suite). On prendra en compte :
- La **fraîcheur** du lead : un lead nouvellement entré a plus de valeur qu'un vieux lead non traité [14] . On peut initialiser un score élevé (par ex. 100) qui décroît avec le temps. Par exemple, diminuer de quelques points chaque jour après l'entrée, ou bien utiliser un système de "lead chaud/froid" (chaud si <24h, tiède si quelques jours, froid si >1 semaine). Plus simplement, la fraîcheur peut être un sous-score calculé = 100 - (nombre d'heures depuis création * un coefficient).
- La **complétude** des informations : un lead ayant tous les champs renseignés (téléphone, email, description détaillée, budget estimé, etc.) sera mieux noté. En effet, *« le lead scoring devient plus précis avec des données complètes »* [15] . Concrètement, on peut ajouter des points si certaines infos clés sont présentes (ex: +20 si numéro de téléphone fourni, +10 si budget indiqué, etc.).
- Potentiellement, d'autres critères à terme : provenance du lead (ex: organique vs partenaire), correspondance du client aux critères (si on cible un type de clientèle idéal), etc. Pour démarrer, on se concentre sur fraîcheur et complétude comme demandé.

Ce score du lead (exprimé sur 100 ou toute échelle choisie) sera stocké dans le champ `Leads.score`. Il pourra être mis à jour en temps réel ou par une tâche planifiée. Par exemple, on peut mettre en place un **Edge Function planifiée quotidiennement** (via la planification Supabase ou un cron externe) pour décrémenter légèrement les scores des leads chaque nuit afin de refléter le vieillissement. Lovable permet de programmer de telles tâches (cf. possibilité de job quotidien à 7h dans la doc) [16] . Alternativement, un calcul "on the fly" de l'âge du lead pourrait se faire dans les requêtes, mais stocker le score pré-calculé est plus simple pour le tri et le filtrage.

- **Score de l'artisan** : vise à évaluer la performance de l'artisan selon deux axes principaux:
- **Réactivité** : à quelle vitesse l'artisan prend en charge ses leads. Un indicateur pourra être le **délai moyen de réponse** (temps entre l'assignation du lead et le passage en « En contact »). On peut aussi mesurer le pourcentage de leads contactés sous 24h. Par exemple, si un artisan contacte 80% de ses leads dans la journée, son score de réactivité sera haut. Ce genre de métrique nécessite de tracer les timestamps (d'où l'importance des champs `date_contact` dans Assignations). Le score pourrait être un nombre ou un label (ex: "Rapide", "Moyen", "Lent") calculé périodiquement. On peut initialiser un score par défaut puis l'ajuster à chaque nouvelle donnée : ex, chaque fois qu'un lead passe « En contact », recalculer la moyenne mobile de temps de réponse.
- **Signature/Conversion** : le taux de transformation des leads en chantiers gagnés. On calculera le ratio nombre de leads « Gagnés » sur nombre total de leads reçus (ou sur un intervalle de

temps donné). Ce taux de signature peut être exprimé en pourcentage. Un artisan qui signe beaucoup de devis aura un score élevé ici.

Pour implémenter cela, on peut ajouter dans la table **Artisans** des champs tels que `total_leads_assignes`, `leads_gagnes`, `taux_conversion` et `delai_moyen_reponse`. Chaque fois qu'une assignation change d'état, une logique (trigger ou edge function) mettra à jour ces compteurs : - À l'ajout d'une assignation (nouveau lead pour artisan), incrémenter `total_leads_assignes` de l'artisan. - Au passage d'un lead en "En contact", enregistrer le délai (now - date_assignation) dans un historique ou recalculer une moyenne pondérée pour `delai_moyen_reponse`. - Au passage en "Gagné" ou "Perdu", incrémenter le compteur correspondant et recalculer `taux_conversion = leads_gagnes / total_leads_assignes * 100`.

On peut choisir de recalculer ces valeurs dynamiquement lors de l'affichage aussi, mais les stocker et mettre à jour sur l'événement offre de meilleures performances pour un tableau de bord. Ces opérations seront implémentées via un **workflow automatisé** de Lovable : possiblement un trigger Supabase (PL/pgSQL) ou plus simplement une Edge Function appelée par l'application lors du changement de statut. Par exemple, lors du clic « Marquer comme gagné », en plus de changer l'état dans Assignations, on appelle une fonction qui incrémente le compteur du profil artisan et met à jour son taux. Lovable.dev permet ce genre d'automatisation « sans écrire de requêtes » en orchestrant les MAJ de tables [17], ou via une fonction backend personnalisée.

Les **scores artisans** pourront être affichés sur le dashboard de l'artisan (pour qu'il suive sa perf) et servir côté plateforme (pour classer les artisans ou attribuer les leads équitablement). Notons que le scoring artisan peut évoluer en complexité (on pourrait intégrer des notes de satisfaction client plus tard, etc.), mais le plan se concentre sur réactivité et signature comme demandé.

5. Génération automatique d'e-mails via Resend

Pour assurer la communication automatique (notifications), nous allons intégrer le service **Resend** dans notre backend Lovable. Resend est une API d'envoi d'e-mails transactionnels moderne, bien adaptée aux besoins de notification de plateforme [18]. Lovable.dev dispose d'une intégration validée pour Resend [19], ce qui facilite son utilisation. L'idée est de déclencher des e-mails à divers moments clés du cycle, sans intervention manuelle, en utilisant soit des Edge Functions qui appellent l'API Resend, soit l'SDK Resend directement.

Les principaux e-mails automatiques à mettre en place : - **Notification de lead reçu (nouveau lead)** – *Destinataire :* l'artisan. Quand un lead est assigné à un artisan (création d'une entrée Assignation), le système envoie immédiatement un e-mail à l'artisan pour l'alerter. Contenu : informations du prospect (nom, ville, travaux demandés) et un appel à action pour se connecter au dashboard et traiter le lead. *Objectif:* assurer que l'artisan est réactif dès qu'un lead arrive. - **Confirmation d'envoi de devis** – *Destinataire :* le client (particulier) et/ou éventuellement l'artisan en copie. Quand l'artisan soumet un devis via la plateforme (upload ou formulaire), un e-mail est envoyé au client pour lui signaler qu'il a reçu un devis de la part de l'artisan (avec éventuellement le devis en pièce jointe ou un lien pour le consulter). On peut utiliser un modèle d'e-mail convivial ("Votre devis pour [Nom du projet] est prêt"). Si le client n'a pas de compte, ce mail peut contenir directement les coordonnées de l'artisan pour le joindre. *Côté artisan*, on peut aussi lui confirmer que le devis a bien été transmis. - **E-mails de relance (follow-up)** – plusieurs scénarios sont possibles : - Relance de l'artisan *vers le prospect* : Par exemple 7 jours après l'envoi du devis, si le statut n'est toujours pas « Gagné » ni « Perdu », on peut rappeler le client ("Nous restons à votre disposition…"). Mais ceci sort un peu du back-end artisan (ce serait plus un outil d'aide commerciale). - Relance de la plateforme *vers l'artisan* : Rappel de traiter ses leads. Par

exemple, si un lead reste en statut « Nouveau » plus de 24h, envoyer un e-mail automatique à l'artisan du style "N'oubliez pas de contacter le prospect X, les leads les plus frais ont le plus de chance d'être convertis". De même, si un devis a été envoyé mais pas de news en X jours, on peut suggérer à l'artisan de relancer le client. Ce type de relance améliore le taux de conversion en maintenant l'artisan actif.

Ces relances devront être réalisées via des tâches planifiées ou des checks réguliers (un job quotidien qui scanne les assignations en retard par exemple). On pourrait utiliser l'intégration de Lovable avec **Make.com** ou **n8n** pour les workflows complexes sans code [20], ou plus simplement écrire une Edge Function planifiée. - **Avertissement documents expirés** – *Destinataire :* l'artisan. Lorsqu'un document important (par ex. assurance, RGE) approche de sa date d'expiration ou a expiré, un e-mail est envoyé pour demander à l'artisan de mettre à jour ce document sur TravauxHub. Ex : "Votre certificat RGE expire dans 15 jours, pensez à le renouveler pour continuer à recevoir des leads." Ce mécanisme nécessite de vérifier régulièrement les dates d'expiration dans la table Documents. On peut planifier, par exemple, un job hebdomadaire qui envoie une alerte si `expiration_date` est dans moins de 30 jours.

Implémentation technique avec Resend : Pour tous ces e-mails, on va recourir à l'API Resend via une fonction backend. Lovable nous permet d'intégrer l'API en toute sécurité : - On stockera la clé API secrète de Resend dans les **Secrets** de Lovable (les variables d'environnement sécurisées injectées dans les Edge Functions) [21]. Cela évite de l'exposer en clair. - On écrira des Edge Functions en TypeScript (déployées sur Supabase Edge Functions) pour appeler les endpoints Resend et envoyer les mails requis. Par exemple, une fonction `sendLeadEmail(artisanId, leadId)` qui récupère les infos nécessaires en base puis utilise le SDK Resend ou une requête HTTP vers Resend pour composer le message. La doc Lovable/Resend suggère d'**utiliser le SDK Resend** pour faciliter l'envoi [22]. - Chaque trigger identifié plus haut (assignation créée, devis créé, etc.) appellera la fonction correspondante. Lovable permet de définir que "lorsqu'un nouveau lead est ajouté… envoyer un email" en quelques étapes [23]. Dans notre cas, on branchera par exemple l'appel de `sendNewLeadEmail` sur l'événement d'insertion dans Assignations (peut-être via un trigger SQL OR via l'application Lovable qui appelle la fonction après création). - Le contenu des e-mails sera rédigé en soignant la personnalisation. On pourra créer des **templates d'e-mail** éventuellement avec React Email (librairie supportée par Resend) pour de jolis designs [24], bien que ce soit optionnel. Pour démarrer, des emails texte/HTML simples suffisent. - Il faudra *vérifier le domaine d'envoi sur Resend* en amont (configuration DNS) [25] pour assurer une bonne délivrabilité (mais cela sort du périmètre Lovable, c'est côté Resend).

En résumé, l'intégration de Resend se fait **via du code backend** : on a besoin de quelques fonctions serverless pour orchestrer ces emails. Lovable facilite cela en autorisant l'appel d'APIs externes depuis les Edge Functions et en offrant un environnement Node.js où on peut utiliser le SDK Resend. Cette approche suit les recommandations de Lovable *« Option 1: utiliser les Edge Functions comme proxy sécurisé pour appeler l'API externe »* [21]. Ainsi, l'équipe pourra adapter le contenu ou ajouter d'autres notifications en modifiant ces fonctions sans refondre tout le système.

6. Suivi de chantier (gestion des projets gagnés)

Lorsqu'un lead est converti en chantier (étape « Gagné »), TravauxHub doit permettre aux artisans de **suivre l'avancement des travaux** jusqu'à leur achèvement. Le back-end va donc gérer des enregistrements de chantiers (table **Chantiers**) et les informations associées.

Création d'un chantier : Comme évoqué, on crée un nouveau record dans **Chantiers** dès qu'une assignation passe à "Gagné". Cela peut se faire automatiquement dans le même workflow que la mise à jour de statut. La fiche chantier est initialisée avec : - Référence au lead/projet et à l'artisan, - Copie du

devis accepté (on peut lier le devis ou stocker son identifiant dans le champ `devis_signee_id`), - Statut initial du chantier, probablement “À démarrer” ou “Planifié” avec une date de début estimée si connue.

Champs de statut du chantier : On définit plusieurs états pour baliser la progression. Par exemple : “Planifié” (en attente de début), “En cours”, “En pause” (si applicable), “Terminé”. L’artisan mettra à jour ce champ selon l’avancement réel. Le champ `date_debut_reelle` sera renseigné quand le chantier commence effectivement, de même `date_fin_reelle` une fois terminé. Ces informations permettent de calculer la durée réelle et de comparer au prévu.

Photos et documents du chantier : Un volet important est de permettre à l’artisan d’**uploader des photos** du chantier (avant/après, ou pendant les travaux) et éventuellement d’autres documents (par ex. un compte-rendu de fin de chantier, des factures, etc.). Grâce à Lovable, on peut intégrer des champs de type file upload reliés à la table Documents pour chaque chantier. Concrètement : - L’artisan, depuis la page de détail du chantier, pourra ajouter une ou plusieurs photos. Chaque upload crée une entrée dans Documents avec `type = photo_chantier` et référence le chantier. Les fichiers sont stockés dans le bucket Supabase configuré, et Lovable gère l’accès (possibilité de rendre les URLs publiques ou restreintes selon besoins). - Idem pour des documents PDF (ex: procès-verbal de réception, attestation de fin de travaux…), on les stocke avec `type = doc_chantier`. - Ces fichiers seront listés sur la page du chantier pour consultation. On peut même imaginer une galerie pour les photos. Tout ceci sans back-end lourd : Lovable s’occupe du storage, il suffira de lier la bonne référence et d’afficher les images (une simple <img> avec l’URL retournée, ou via composant image).

Suivi et mise à jour : Sur le dashboard, l’artisan aura la liste de ses chantiers en cours (issus de ses leads gagnés). Il peut entrer dans un chantier pour voir/modifier : - Le statut (avec un menu déroulant pour passer de “En cours” à “Terminé” par ex). - Les dates (il pourrait ajuster la date de fin prévue si retard, etc.). - Ajouter des **notes ou commentaires** sur le chantier (qu’on pourrait stocker soit dans la table Chantiers comme champ texte de log, soit utiliser la table Messages pour journaliser les notes internes liées au chantier). - Consulter le **devis signé** du projet. S’il a été uploadé, on affiche le PDF; sinon on montre les infos du devis enregistré. - Consulter/ajouter des photos & docs comme vu ci-dessus.

Chaque modification importante du chantier peut créer un **Événement** dans la timeline (ex: “Chantier marqué Terminé le 12/12/2025”). De plus, on peut prévoir d’**alerter le client par e-mail** à certaines étapes, si leur email est connu : par exemple un mail “Votre chantier X est terminé, merci de votre confiance”. Ce n’est pas demandé explicitement, mais c’est une fonctionnalité potentielle. Cela se ferait via Resend comme les autres.

Techniquement, la plupart des fonctions de suivi de chantier pourront être réalisées sans code lourd : c’est essentiellement de la gestion de données (CRUD) sur la table Chantiers et de la **mise à jour de statut**. Lovable permettra de créer une page “ChantierDetail” avec les champs modifiables et l’upload de fichiers (puisqu’il gère le storage). Aucune difficulté majeure n’est prévue ici, il faudra juste veiller à restreindre l’accès : seul l’artisan assigné (et éventuellement les admins) doivent voir le chantier correspondant. On appliquera donc des règles de sécurité (par ex. row-level security dans Supabase pour que `chantier.artisan_id == auth.uid` ou équivalent). L’intégration de la sécurité est à configurer, mais Lovable facilite ce genre de configuration lors de l’ajout de l’auth [6].

7. Carte des artisans géolocalisés

Afin de visualiser la **répartition géographique des artisans** et de faciliter le matching local, on va inclure une carte affichant les artisans selon leur zone d'intervention. Du point de vue back-end, il faut d'abord assurer qu'on stocke bien les données de localisation nécessaires, puis exposer ces données à un composant cartographique.

Données géo à stocker : Pour chaque artisan, on a déjà prévu de stocker la ville (ou adresse) principale et un rayon (km). Il serait utile de convertir la ville en coordonnées GPS (latitude/longitude). Deux approches possibles : - **Au moment de l'inscription**, faire un appel à une API de géocodage (ex: Mapbox Geocoding API ou Google Geocoding) pour traduire l'adresse/ville en lat & long. On peut intégrer cette étape via une Edge Function qui utilise l'API (Lovable permet d'utiliser n'importe quelle API externe avec un peu de configuration et des clés d'API sécurisées [26] [21]). Par exemple, appeler Mapbox Geocoding avec la ville, récupérer lat/long et les stocker dans `artisan.latitude`, `artisan.longitude`. - Ou demander directement à l'artisan de pointer son adresse sur une carte lors de l'onboarding, mais c'est plus complexe en UI. Le plus simple est la méthode ci-dessus pour ne pas alourdir l'expérience utilisateur.

Avec les coordonnées disponibles par artisan, et le rayon, on peut calculer si un point (lead) est dans le rayon (via la formule de la distance entre deux coordonnées) dans les Edge Functions d'assignation de leads. Cela rendra le matching plus précis qu'une simple comparaison de ville. Supabase (PostGIS) peut aussi gérer ce calcul via une requête géospatiale si configuré, mais on peut tout aussi bien le coder.

Affichage d'une carte : Lovable.dev n'a pas de composant map natif connu, mais on peut intégrer une librairie. Par exemple, **Mapbox** est mentionné parmi les intégrations possibles [27] . On pourrait donc inclure Mapbox GL JS dans le front-end : - Ajouter la librairie Mapbox (via script or package) et une API Key Mapbox (clé publique) dans les settings. - Créer une composant ou page "Carte des Artisans" (accessible aux admins, ou même aux clients pour voir les pros près de chez eux, selon le besoin). - Ce composant appelle la base pour obtenir la liste des artisans avec leurs lat/long et rayon. Ensuite, en JS, on initialise la carte centrée sur une région (par ex. la France ou la zone couverte) et on place des **marqueurs** pour chaque artisan. - On peut, si utile, afficher un cercle autour du marqueur représentant le rayon d'intervention. Mapbox GL JS permet de dessiner des cercles ou polygones facilement. - Pour plus d'interactivité, cliquer un marqueur pourrait afficher le nom de l'artisan et ses infos (métier, etc.).

Côté backend, ceci n'ajoute pas de table supplémentaire, c'est juste de l'utilisation des données existantes. Il faudra simplement prévoir une **API ou une requête sécurisée** pour obtenir la liste des artisans avec coordonnées. Dans Lovable, on peut créer un **Edge Function ou un RPC Supabase** qui retourne ces données, ou directement interroger la table `Artisans` depuis le front (puisque c'est notre propre app, on peut peut-être exposer directement avec les droits admin ou via un filtre). Si on veut éviter d'exposer tous les artisans à tout le monde, on réserve la carte aux admins ou on n'affiche que de façon agrégée.

Notons qu'une **intégration avec Mapbox** s'aligne avec les capacités de Lovable à intégrer des APIs externes en fournissant la doc et la clé API [26] . Une fois Mapbox intégré, Lovable (via l'AI) peut même aider à générer le code pour afficher une map avec des points. Cette partie front est importante pour l'UX, mais comme demandé nous nous assurons que le backend fournit bien les données nécessaires (coords, rayon) et peut éventuellement calculer des distances.

En complément, la géolocalisation sert aussi dans le **matching**: quand un lead arrive avec une adresse, on peut calculer en backend quels artisans sont dans un rayon donné. Une optimisation future pourrait être d'utiliser une **requête PostGIS** pour trouver tous artisans où `ST_DWithin(artisan.geom, lead.geom, artisan.rayon)` est vrai, mais au début un simple parcours calculé en JS/TS suffira vu le volume modeste attendu.

8. Automatisations dans Lovable.dev (workflows, règles de gestion)

Dans cette section, nous récapitulons **comment implémenter la logique automatisable** décrite plus haut à l'aide des outils Lovable.dev. L'objectif est de minimiser le code manuel en profitant des workflows configurables et de l'AI de Lovable, tout en assurant la fiabilité via des fonctions serveur pour les tâches critiques.

- **Workflows sans code (Lovable UI)** : Beaucoup de comportements peuvent être définis via la logique de l'application générée par Lovable. Par exemple, on peut configurer des actions déclenchées par des événements utilisateurs:
- Lorsqu'un artisan clique "Marquer comme contacté" sur un lead, l'interface peut directement mettre à jour le champ `Assignations.statut = En contact` et enregistrer `date_contact = now()`. Ceci ne requiert pas de code serveur explicite, c'est une simple opération de mise à jour de ligne que Lovable sait faire via son intégration Supabase [17].
- De même, soumettre un formulaire de devis peut créer un enregistrement dans la table Devis et mettre à jour le statut de l'assignation en une seule suite d'actions. On peut le décrire à Lovable en langage naturel et l'outil générera la séquence appropriée.
- L'upload de fichiers (documents) se fait sans code via les composants d'upload reliés au Storage Lovable [3].

En résumé, tout ce qui est **création ou modification de données simple** (CRUD) peut être réalisé via Lovable **sans écrire de code** en le décrivant dans le chat de construction ou via les options de l'UI (par exemple, "on form submit, insert into Devis and update Assignations.status").

- **Règles de scoring et calculs** : Pour le scoring lead/artisan, on peut implémenter certaines logiques via des **colonnes calculées** ou des vues dans la base, mais c'est parfois limité (surtout pour la notion de temps réel). Au lieu de surcharger la DB de triggers SQL, on peut utiliser les Edge Functions:
- Une *Edge Function* `assignLeadToArtisans(leadId)` pour la distribution de leads. Elle va chercher les artisans éligibles et créer les assignations. Cette fonction sera appelée soit directement quand un lead est créé (via un hook) soit par un admin via un bouton "Assigner". **Lovable** peut être invité à écrire cette fonction en pseudo-code puis en TS, en tenant compte de la distance géographique (avec soit un calcul custom, soit en utilisant par exemple un appel Mapbox Matrix API pour obtenir des distances routières – ça serait un plus).
- Une fonction `updateScoresOnStatusChange(assignmentId, newStatus)` qui, déclenchée lors d'un changement de statut, va mettre à jour les scores concernés. Par exemple, si `newStatus == "En contact"`, on calcule le délai et on stocke/recalcule la réactivité de l'artisan; si `== "Gagné"`, on incrémente les compteurs et scores de conversion. Cette fonction peut être branchée soit via un trigger Supabase sur la table Assignations (AFTER UPDATE) ou appelée explicitement dans le front-end logic *après* la mise à jour de statut. Dans Lovable, on peut orchestrer ça en disant *"après le changement de statut, appeler la fonction X"*.
- Calcul du score des leads à la création : on peut utiliser un **trigger** sur insertion de Lead qui calcule le score et l'écrit. Ce trigger peut être écrit en PL/pgSQL ou comme ci-dessus en appelant

une Edge Function. Lovable facilite la création de fonctions backend, on pourrait donc faire calculer le score par une fonction JS (plus facile à maintenir qu'une fonction SQL complexe).

À noter qu'on pourrait tenter d'utiliser les fonctionnalités *AI de Lovable* pour, par exemple, analyser la description du projet et en déduire un score d'intérêt, mais c'est optionnel et hors scope.

- **Envoi des e-mails Resend (automatisation)** : Comme détaillé, on utilisera des Edge Functions comme pont. Par exemple:
- Une fonction `sendEmailLeadAssigned(artisanId, leadId)` qui envoie le mail de nouveau lead. À configurer pour se déclencher sur un nouvel enregistrement Assignations (Supabase propose les "DB Webhooks" ou triggers vers une HTTP function, ce qui peut être couplé à l'Edge Function).
- Une fonction `sendEmailDocExpiry(artisanId, docId)` appelée par un job planifié quotidiennement qui parcourt les docs expirant bientôt.

Lovable documente l'envoi d'emails via Edge Functions dans son guide CRM [23] et via l'intégration Resend [28]. Par exemple, *« envoyer un email de confirmation à chaque soumission de formulaire »* est un cas d'usage présenté [28], que l'on adapte ici à nos événements métier.

- **Fonctions planifiées** : Pour les relances et vérifications d'expiration, Lovable n'a pas de scheduler UI direct, mais on peut utiliser Supabase Cron (si disponible) ou un service externe (Make, Zapier) qui frappe une URL d'Edge Function à intervalle régulier. Par exemple, configurer un scénario Make qui appelle `sendReminders()` tous les jours à 8h. L'avantage d'un service comme Make.com, c'est sa facilité à orchestrer sans code et c'est intégré avec Lovable [20]. On pourrait donc sans trop coder mettre en place : à 8h, requête sur Supabase (via lovable integration ou edge function) pour sélectionner les leads non contactés >24h, puis envoi de mail via Resend. Ce mix d'outils peut éviter de développer une mécanique de cron maison.

- **Sécurité et règles d'accès** : Lovable gère l'authentification et on peut configurer des **Row-Level Security** sur les tables via Supabase si nécessaire (ex: un artisan ne peut SELECT que ses leads/chantiers). Par défaut, Lovable propose que chaque table ait une policy en fonction de l'utilisateur courant. Il faudra les adapter : par ex, la table Leads pourrait être non accessible directement aux artisans (puisqu'ils passent par Assignations), la table Assignations accessible en lecture filtrée sur artisan_id = uid, etc. On exploitera l'interface Lovable ou Supabase Studio pour définir ces règles, sans avoir à coder de système d'autorisation from scratch.

En somme, **Lovable.dev permet d'automatiser la majorité des flux** décrits grâce à ses intégrations étroites avec Supabase (BD, Auth, Storage) et la possibilité de coder des Edge Functions pour les besoins spécifiques. L'équipe pourra créer ces workflows par des instructions en langage naturel via l'interface Lovable, par exemple en détaillant : *« Quand un nouvel lead est inséré dans la table Leads, créer des assignations pour les artisans correspondants et envoyer un email via Resend à chacun »*, et l'AI de Lovable générera le code ou les étapes nécessaires. Bien sûr, il faudra tester et ajuster, mais c'est **techniquement faisable dans le cadre de Lovable**. Les cas d'usage courants (email, CRUD, authentification) sont précisément ceux mis en avant par la plateforme [23] [28].

9. Fonctions no-code vs. code personnalisé / tâches manuelles

Pour clarifier le périmètre de ce qui peut être géré **sans coder** (via configuration ou fonctionnalités standard Lovable) et ce qui nécessitera du développement supplémentaire ou des actions humaines, voici un récapitulatif :

Ce qu'on peut réaliser sans code (ou avec minimum de code) :

- **Authentification utilisateurs** : 100% géré par Lovable/Supabase out-of-the-box (pages de login/signup, récupération de mot de passe) [2] .
- **Création des tables et relations** : via les prompts Lovable, on décrit les entités et le système génère le schéma [1] . Pas besoin d'écrire du SQL manuel pour la base initiale.
- **Formulaires et CRUD de base** : Lovable permet de générer des formulaires d'inscription artisan, de mise à jour de profil, de soumission de devis, etc. L'insertion/mise à jour en base se fait via des actions configurées (que Lovable peut produire en code, mais sans que l'on ait à l'écrire nous-mêmes).
- **Upload et gestion de fichiers** : aucun dev back nécessaire, on utilise le composant d'upload relié à Supabase Storage [3] . Le paramétrage des buckets et l'affichage des fichiers (via URLs signées ou publiques) est possible par configuration.
- **Tableaux de bord et calculs simples** : Pour afficher les KPI, si on stocke déjà les valeurs calculées (taux, moyennes), il suffit de les lire et éventuellement de faire des calculs simples en JavaScript dans le front-end (que Lovable peut intégrer). Par exemple, calculer un pourcentage pour affichage peut être fait côté front sans déclencher de code serveur. Les composants graphiques (graphiques, cartes) peuvent être ajoutés en incluant des librairies comme Highcharts ou Mapbox, ce qui est supporté nativement par Lovable [7] [27] sans avoir à héberger nous-mêmes ces services.
- **Transitions d'états usuelles** : Les changements de statut par action utilisateur (drag & drop ou bouton) peuvent être mis en œuvre via la logique front (modifier un champ). Lovable peut générer un code d'interface qui met à jour la donnée et éventuellement appelle une fonction (si on le lui dit). Donc par exemple, passer manuel de "Nouveau" à "En contact" ne requiert pas d'écrire de code backend.
- **Envoi d'e-mail basique** : Lovable peut aider à intégrer rapidement Resend. En suivant le tutoriel, on voit qu'en moins d'une heure on peut avoir un CRM avec envoi d'emails automatisés [29] . Essentiellement, il s'agit de quelques lignes de code d'intégration du SDK Resend – que Lovable peut proposer d'écrire si on fournit la doc – et l'essentiel du setup (clé API, domaine, etc.) est assez direct [30] . Donc envoyer un email à l'insertion d'un lead ou à la soumission d'un formulaire peut être réalisé en grande partie via l'assistant AI de Lovable (il va générer l'Edge Function correspondante).
- **Intégrations externes** : L'usage de services comme Make.com pour planifier des workflows évite d'avoir à coder un scheduler. De même, si on voulait brancher Slack ou autre, Lovable a des intégrations prêtes. Pour notre cas, Mapbox par exemple s'intègre bien si on fournit la clé API et la doc, l'AI pourra nous aider à afficher la carte [27] . Pas besoin d'écrire un module complet de map, on réutilise ce qui existe.

En bref, **Lovable excelle pour tout ce qui est standard** : formulaires, routes sécurisées, stockage, petites fonctions de glue (Edge Functions) qu'il peut générer sur demande. On l'utilisera autant que possible pour gagner du temps.

Ce qui nécessite du code avancé ou une intervention manuelle :

- **Logique de matching géographique et calculs complexes** : La distribution des leads aux artisans en fonction de la distance ou la gestion d'un score qui décroit avec le temps sont des logiques spécifiques. Bien que faisables avec Lovable, cela revient à lui faire écrire du code sur mesure (Edge Function ou trigger SQL). Il faudra sans doute peaufiner ce code manuellement pour s'assurer qu'il est correct (l'AI aide, mais peut "halluciner" comme tout LLM). Donc c'est du développement (en TS ou SQL), même s'il est accéléré par l'outil.
- **Scoring avancé** : Notre scoring de base est simple. Si on voulait un *lead scoring* plus sophistiqué (par ex un modèle prédictif qui score en fonction de données comportementales ou un scoring

artisan qui intègre la satisfaction client), cela sortirait du cadre no-code. On devrait coder l'algorithme ou intégrer un service d'IA. Par exemple, un scoring dynamique avec Machine Learning nécessiterait d'entraîner un modèle et d'appeler une API ML – clairement du développement supplémentaire.

- **Modération et vérification manuelle** : La vérification des documents (RGE, assurance) est un processus humain. On peut faciliter la tâche avec l'UI (ex : une page admin listant les docs à valider, avec un bouton "Valider" qui met à jour le statut), mais c'est l'équipe TravauxHub qui devra effectuer cette action. Aucune automatisation ne peut remplacer le jugement humain ici (sauf à intégrer une API gouvernementale pour vérifier les RGE par SIRET, ce qui serait possible mais complexe). Donc la modération reste **manuelle**.
- **Gestion des contenus ou support client** : Si l'on modère aussi les messages échangés ou si on gère des litiges, ce ne sera pas automatisé. On n'a pas inclus de table pour litiges ou évaluations, mais à prévoir en extension – de toute façon, leur traitement serait en partie manuel.
- **Exports de données** : Si l'équipe TravauxHub souhaite exporter des données (CSV des leads, synthèse mensuelle), cela pourrait être partiellement automatisé (Edge Function qui génère un CSV et l'envoie par email, par ex). Cependant, c'est typiquement une tâche qu'on peut faire manuellement au début (via un accès admin à la base ou en utilisant l'interface Supabase). Un export en un clic pourrait être implémenté plus tard avec du code. Donc initialement, considérez l'export comme **manuel** (ou à configurer dans Supabase Studio).
- **Notifications push ou SMS** : Non évoquées dans le plan, mais si un jour on veut ajouter du SMS (via Twilio) ou des notifications push mobile, ce sont d'autres intégrations à coder/configurer (Lovable a Twilio en intégré toutefois [31]). Ce n'est pas prioritaire ici.
- **Optimisations et scalabilité** : À mesure que la plateforme grandit, on devra peut-être optimiser certaines requêtes ou fonctions (index, etc.). Lovable gère l'infrastructure Supabase, mais des ajustements manuels pourront être nécessaires si les volumes explosent. C'est en dehors du no-code strict.

En synthèse, **les règles métier spécifiques et les processus de qualité** (matching, scoring évolué, vérifications) requièrent l'écriture de fonctions backend sur mesure et/ou l'intervention de l'équipe. Néanmoins, *le cadre Lovable.dev couvre 80% du besoin sans code*, ce qui permet à l'équipe de se concentrer sur ces 20% restants complexes.

10. Conclusion

Ce plan détaillé présente une **architecture back-end cohérente et réalisable sur Lovable.dev** pour la plateforme TravauxHub. En combinant les capacités no-code de Lovable (authentification intégrée, base de données Supabase, stockage de fichiers, intégrations prêtes à l'emploi) avec des **Edge Functions** ciblées pour la logique métier (scoring, assignation, notifications par e-mail, etc.), l'équipe pourra mettre en place un CRM complet pour les artisans sans repartir de zéro. Chaque composant – gestion des artisans, pipeline de leads, suivi des chantiers, carte interactive, automatisations – s'insère dans le cadre technique de Lovable et Supabase, comme l'illustrent les références utilisées (ex: envoi d'e-mails [22], transitions pipeline auto [10], etc.).

Le plan souligne également les points nécessitant une attention particulière ou du travail manuel (par ex. validation des documents, configuration des clés API pour Resend/Mapbox, tests des triggers automatiques). Ces éléments devront être pris en compte lors du déploiement.

Prochaine étape : suivre ce plan pour configurer les tables dans Lovable, implémenter les workflows d'inscription et pipeline, puis coder les quelques fonctions backend pour le scoring et l'envoi d'e-mails. En procédant itérativement (d'abord la base artisan/leads, puis ajout du scoring, puis du suivi chantiers, etc.), l'équipe pourra adapter au fur et à mesure en fonction des retours des utilisateurs. Ce plan sert de

feuille de route structurée pour aboutir à un back-end opérationnel, évolutif et aligné sur les besoins métier de TravauxHub, le tout réalisable dans l'environnement Lovable.dev avec un minimum de friction technique.

---

1 2 3 4 5 12 13 16 17 23 Lovable Cloud - Lovable Documentation
https://docs.lovable.dev/features/cloud

6 8 18 22 24 25 28 29 30 Resend integration - Lovable Documentation
https://docs.lovable.dev/integrations/resend

7 19 20 21 26 27 31 Introduction - Lovable Documentation
https://docs.lovable.dev/integrations/introduction

9 10 11 Introduction au CRM | OpenFire
https://documentation.openfire.fr/knowsystem/introduction-au-crm-118

14 Lead vs. Prospect : définition et processus de qualification
https://www.infopro-digital-media.fr/lead-et-prospect-comprendre-les-differences-pour-optimiser-sa-prospection/

15 Enrichir base de données B2B : méthodes et outils efficaces
https://datapult.ai/guides/enrichir-base-donnees-b2b-methodes-outils/