1. Problem Statement

Finding reliable local labor for short-term tasks remains a fragmented and trust-deficient process:

- **Homeowners** need help with moving, cleaning, repairs, gardening, etc.
- **Small businesses** require temporary staff for events, inventory, or projects.
- **Workers** (students, freelancers, skilled tradespeople) seek flexible, fairly compensated opportunities.

Existing solutions are either:

- General freelance platforms (Upwork, Fiverr) – not suited for physical, location-based labor.
- Classified ads (Craigslist, local Facebook groups) – lack verification, safety, and structured workflows.

A dedicated labor marketplace must handle:

- Worker verification and trust.
- Task posting and discovery with location awareness.
- Scheduling and conflict prevention.
- Secure payments with escrow.
- Dispute resolution and ratings.

Goal: Build a full-featured ASP.NET MVC web application that connects task posters and workers in a safe, efficient, and scalable manner.

---

2. Core Features

Must-have

Feature	Description
User Roles	Task Poster (client) and Worker (laborer) – registration and login.
Task Posting	Poster creates a task: category, description, location (map), date/time, budget type

Feature	Description
	(fixed/hourly), budget amount.
Worker Discovery	Workers can search for tasks by location (radius), category, date, and budget. Results r by proximity, rating, response time.
Task Application	Workers apply to tasks with a message and proposed rate. Posters can invite specific v
Booking & Scheduling	When both parties agree, a booking is created. Workers cannot accept overlapping boo (concurrency check).
In-app Messaging	Real-time chat between poster and worker for coordination.
Payments (Escrow)	Client funds the escrow before work starts. Payment released after task completion (b parties confirm) or dispute resolution.
Ratings & Reviews	After each completed booking, both parties rate each other (1–5 stars) with optional c
Basic Verification	Email and phone verification. Optional ID upload (stored securely).

Optional (advanced)

- Background check integration (via third-party API).
- Insurance add-on for high-risk tasks (e.g., tree removal).
- Recurring bookings (weekly cleaning, monthly lawn care).
- Urgent tasks with surge pricing multiplier.
- Worker availability calendar (block out times).
- Admin dashboard for dispute resolution and platform monitoring.
- Geolocation check-in/out for hourly tasks (using mobile GPS).

---

3. Technology Stack

Layer	Technology
Frontend	ASP.NET MVC 5 (Razor views), Bootstrap 5, jQuery, SignalR (real-time), Leaflet.js (ma
Backend	.NET Framework 4.8 / .NET 6+ (if using modern ASP.NET Core MVC)
Database	SQL Server (or SQL Azure) with Entity Framework 6 / EF Core
Caching	Redis (optional, for session state and real-time data)
Search	SQL Server Full-Text Search + spatial indexes, optionally Elasticsearch
Background Jobs	Hangfire (for reminders, payment release)
Payment Gateway	Stripe Connect (handles escrow and transfers)
Real-time	SignalR (for chat and live notifications)
Authentication	ASP.NET Identity with roles (Poster, Worker)
Hosting	Azure App Service / IIS

---

4. Architecture Overview

The application follows the **Model-View-Controller** pattern with an n-tier architecture:

text

```
┌──────────────────────────────────────────────────────────────
──┐
│                  Presentation Layer                  │
│          Razor Views + JavaScript (jQuery, SignalR)          │
```

```
└────────────────────────────────────────┬───────────────────────────────────
───┘
                │
┌────────────────────────────────────────▼──────────────────────────────────
────┐
│           Controllers Layer               │
│ AccountController, TaskController, BookingController, etc. │
└────────────────────────────────────────┬───────────────────────────────────
───┘
                │
┌────────────────────────────────────────▼──────────────────────────────────
────┐
│           Business Logic Layer              │
│    Services (TaskService, BookingService, Payment)     │
└────────────────────────────────────────┬───────────────────────────────────
───┘
                │
┌────────────────────────────────────────▼──────────────────────────────────
────┐
│          Data Access Layer (EF)             │
│         Repositories, DbContext, Entities          │
└───────────────────────────────────────────────────────────────────────────
───┘
```

Key Design Decisions

- **Separation of Concerns**: Controllers thin, business logic in services.
- **Repository Pattern**: Abstracts data access, easier unit testing.
- **Dependency Injection**: Using Unity/Autofac or built-in DI in .NET Core.
- **Real-time Communication**: SignalR hubs for chat and notifications.

- **Background Processing**: Hangfire for scheduled jobs (e.g., release payment after 24h if no dispute).

---

5. Database Schema (Key Entities)

Users (AspNetUsers – extended)

Column	Type	Description
Id	string	
FirstName	string	
LastName	string	
PhoneNumber	string	
PhoneConfirmed	bool	
IDVerified	bool	Whether ID uploaded and verified
AverageRating	decimal	Computed from reviews
Location	string	Last known location (optional)
Coountry	string	
CreatedAt	Datetime	

There is two types of users admin and client which may be worker or poster

Tasks

Column	Type	Description
Id	int (PK)	
PosterId	string	FK to AspNetUsers
Category	int	Enum: Cleaning, Moving, Repair, etc.
Title	string	Short description
Description	string	Detailed description
Location	geography	SQL Server spatial type (point)
Address	string	Human-readable address
ScheduledStart	datetime	
ScheduledEnd	datetime	
BudgetType	int	0 = Fixed, 1 = Hourly
BudgetAmount	decimal	
Status	int	Open, Assigned, InProgress, Completed, Cancelled, Disputed
CreatedAt	datetime	
UpdatedAt	datetime	

TaskApplications

Column	Type	Description
Id	int (PK)	
TaskId	int	FK to Tasks
WorkerId	string	FK to AspNetUsers
ProposedRate	decimal	Worker’s proposed rate
Message	nvarchar(max)	Optional cover message
Status	int	Pending, Accepted, Rejected
CreatedAt	datetime	

Bookings

Column	Type	Description
Id	int (PK)	
TaskId	int	FK to Tasks
WorkerId	string	FK to AspNetUsers
AgreedRate	decimal	Final rate after negotiation
StartTime	datetime	Actual start (nullable)
EndTime	datetime	Actual end (nullable)

Column	Type	Description
Status	int	Scheduled, InProgress, Completed, Cancelled, Disputed
Version	rowversion	Concurrency token
CreatedAt	datetime	

Payments

Column	Type	Description
Id	int (PK)	
BookingId	int	FK to Bookings
Amount	decimal	Amount held in escrow
Status	int	Held, Released, Refunded, PartiallyRefunded
StripePaymentIntentId	string	Reference to Stripe
CreatedAt	datetime	
ReleasedAt	datetime	Nullable

Messages

Column	Type	Description
Id	int (PK)	
BookingId	int	FK to Bookings (chat per booking)

Column	Type	Description
SenderId	string	FK to AspNetUsers
Content	nvarchar(max)	
SentAt	datetime	
IsRead	bool	

Reviews

Column	Type	Description
Id	int (PK)	
BookingId	int	FK to Bookings (unique)
ReviewerId	string	FK to AspNetUsers
RevieweeId	string	FK to AspNetUsers
Rating	int	1–5
Comment	nvarchar(500)	
CreatedAt	datetime	

Disputes

Column	Type	Description
Id	int (PK)	

Column	Type	Description
BookingId	int	FK to Bookings
RaisedBy	string	FK to AspNetUsers
Reason	nvarchar(max)	
Status	int	Open, UnderReview, Resolved
Resolution	nvarchar(max)	Admin notes
CreatedAt	datetime	
ResolvedAt	datetime	

---

6. Critical Business Rules & Challenges

6.1. Concurrency & Double-Booking Prevention

- **Rule**: A worker cannot have two accepted bookings with overlapping time windows.
- **Implementation**:
  - Use database serializable transactions when accepting an application.
  - Check existing bookings for the worker: (Start < new.End) and (End > new.Start).
  - Use optimistic concurrency with a rowversion column on the worker's schedule (or on Bookings table) to detect conflicts.

6.2. Escrow Payment Flow

- **Rule**: Client must fund the escrow before work starts. Payment released only after:
  - Both parties mark task as completed, OR
  - Dispute resolution concludes.

- **Implementation**:
    - Use Stripe Connect to create a PaymentIntent with capture_method: manual (holds funds).
    - After completion, call capture to release.
    - Handle webhooks for payment failures; use idempotency keys.

6.3. Cancellation & No-Show Penalties

- **Rule**:
    - Client cancels < 2 hours before start: forfeit 50% to worker.
    - Worker no-show: rating penalty, temporary account suspension after repeated offenses.
- **Implementation**:
    - Hangfire job runs every 15 minutes, checks tasks starting within 2 hours.
    - If cancellation occurs, trigger refund/penalty logic.
    - For no-show: worker can mark arrived via GPS check-in; if not marked, auto-flag after start time +30 min.

6.4. Rating Integrity

- **Rule**: Each booking can be rated once by each party. Ratings are visible only after both parties have rated or 14 days have passed.
- **Challenge**: Prevent fake ratings (e.g., friends boosting each other). Use rate limiting and device fingerprinting. In future, ML anomaly detection.

6.5. Verification Tiers

- **Rule**: Unverified workers (email/phone only) can only apply to tasks ≤ $100. Verified (ID uploaded) can apply to all tasks.
- **Implementation**: Check User.IDVerified flag during application submission.

6.6. Dispute Resolution

- **Rule**:
    - Either party can raise dispute within 48h of completion.
    - System freezes payment, both upload evidence (via form).

    - Admin reviews; if no resolution in 7 days, escalate to arbitration (or auto-split 50/50).
- **Implementation**:
    - State machine pattern in Dispute entity with status transitions.
    - Hangfire job escalates unresolved disputes after 7 days.

6.7. Search Ranking Algorithm

- **Rule**: Tasks listed for workers should be ranked by:
    - Distance (40%)
    - Poster's average rating (20%)
    - Task urgency (20%)
    - Budget (20%)
- **Implementation**: Use SQL Server spatial functions (STDistance) combined with weighted scoring. Cache results for performance.

6.8. Surge Pricing for Urgent Tasks

- **Rule**: Tasks starting within 2 hours get 1.5× multiplier on worker's hourly rate.
- **Implementation**: In task creation, calculate effectiveBudget = (IsUrgent ? Budget * 1.5 : Budget). Display to workers with surge tag.

---

7. Key Implementation Details (MVC Specific)

7.1. Controllers & Areas

- **AccountController** – registration, login, profile management.
- **TaskController** – CRUD for tasks, list/search views.
- **ApplicationController** – handle applications (apply, accept, reject).
- **BookingController** – manage bookings (start, complete, cancel).
- **PaymentController** – handle escrow funding, webhooks.
- **MessageController** – chat interface.
- **AdminArea** – separate area for admin functions (disputes, user management).

7.2. Views & Layout

- Responsive Bootstrap 5 layout with navigation depending on role.
- **Task List View** – cards with map preview, filters (category, radius, date).
- **Task Detail View** – full description, apply button, worker list (if poster), chat box.
- **Worker Profile View** – ratings, past reviews, verification badge.
- **Booking Dashboard** – list of upcoming/past bookings for current user.
- **Checkout View** – payment form (Stripe Elements) for client to fund escrow.

7.3. Real-time Chat with SignalR

- Create a ChatHub that manages connections to booking-specific groups.
- When a user opens a booking detail page, they join a group named "booking-{bookingId}".
- Messages are saved to DB via AJAX POST, then broadcast to the group.
- Display unread message counts using SignalR notifications.

7.4. Background Jobs with Hangfire

- **Recurring Job**: Every hour, check for tasks starting soon and send reminders.
- **Fire-and-forget**: When booking is completed, schedule a job to auto-release payment after 24h if no dispute raised.
- **Delayed Job**: If a worker doesn't check in within 30 min of start, auto-flag as no-show.

7.5. Security & Validation

- **Input validation**: Both client-side (jQuery validation) and server-side (Data Annotations, FluentValidation).
- **Anti-forgery tokens** on all POST forms.
- **Authorization**: [Authorize(Roles = "Poster")] etc.
- **Payment security**: Stripe tokens, never handle raw card data.

7.6. Search Implementation

Using SQL Server Full-Text Search and spatial indexes:

sql

```
CREATE SPATIAL INDEX SIndx_Tasks_Location ON Tasks(Location);
```

In LINQ (Entity Framework):

csharp

```
var nearbyTasks = context.Tasks
  .Where(t => t.Location.Distance(userLocation) <= maxDistance)
  .OrderBy(t => t.Location.Distance(userLocation))
  .Select(t => new TaskListItem
  {
    Task = t,
    Distance = t.Location.Distance(userLocation)
  });
```

Ranking with weighted scoring can be done in SQL or in memory after fetching.

7.7. Payment Integration (Stripe Connect)

- Create Stripe accounts for workers (express or custom).
- When client pays, create a PaymentIntent with application_fee_amount (platform fee).
- Funds are held in the platform's Stripe account until release.
- After task completion, call capture to transfer funds to worker's Stripe account.

7.8. Error Handling & Logging

- Global error handling via Application_Error or middleware.
- Log to database, file, or Application Insights.
- User-friendly error pages.