**Especificação do Software: Jogo da Velha com Aprendizado**

**1. Objetivo**

Desenvolver um jogo da velha interativo onde a inteligência artificial (IA) aprende com base em padrões de tabuleiro armazenados em um banco de dados SQL Server. A IA consulta o banco para decidir jogadas, priorizando: (1) vitória imediata, (2) bloqueio de vitória do oponente, (3) jogadas com maior probabilidade de vitória ou empate com base em dados históricos, ou (4) jogadas aleatórias se não houver dados. O sistema será implementado com backend em C# (ASP.NET Web API), frontend em AngularJS (1.x), e banco de dados SQL Server, garantindo uma experiência fluida e aprendizado contínuo da IA.

**2. Requisitos Funcionais**

**2.1. Interface do Usuário**

* **Tabuleiro 3x3**: Exibir um grid 3x3 interativo onde o usuário joga como 'X' e a IA como 'O'.
* **Jogadas do Usuário**: Permitir que o usuário clique em uma célula vazia para marcar 'X'.
* **Resposta da IA**: Após cada jogada do usuário, a IA responde automaticamente com uma jogada ('O') em menos de 1 segundo.
* **Exibição de Resultado**: Mostrar o resultado do jogo (vitória, empate, derrota) ao final, com mensagem clara (ex: "Você venceu!", "Empate!", "IA venceu!").
* **Reinício**: Incluir um botão "Reiniciar" para limpar o tabuleiro e iniciar um novo jogo.
* **Feedback Visual**: Destacar células clicadas (ex: fundo colorido para 'X' e 'O') e desabilitar interações em células ocupadas ou após o fim do jogo.
* **Histórico Simples**: Exibir o número de jogos jogados e resultados (vitórias, empates, derrotas) na sessão atual.

**2.2. Lógica da IA**

* **Consulta ao Banco**: Para cada jogada, consultar a tabela BoardStates para encontrar padrões de tabuleiro semelhantes ao estado atual.
* **Prioridade de Jogadas**:
  1. **Vitória Imediata**: Escolher a jogada que resulta em vitória para 'O' (três 'O' em linha, coluna ou diagonal).
  2. **Bloqueio do Oponente**: Escolher a jogada que impede uma vitória imediata de 'X'.
  3. **Jogada Baseada em Dados**: Selecionar a jogada com maior frequência de resultados positivos ('Win' ou 'Draw') no banco, ordenada por Frequency.
  4. **Jogada Aleatória**: Se não houver dados no banco para o estado atual, escolher uma posição vazia aleatoriamente.
* **Aprendizado**:
  1. Armazenar cada estado do tabuleiro, jogada realizada, e resultado no banco.
  2. Incrementar a frequência (Frequency) de estados existentes em vez de criar duplicatas.
  3. Registrar o resultado final do jogo ('Win', 'Draw', 'Loss') na tabela GameResults.

**2.3. Banco de Dados**

* **Estrutura**:
  + **Tabela BoardStates**:
    - Armazena estados do tabuleiro, jogadas, resultados, e frequência.
    - Campos: Id (PK), Board (VARCHAR(9), ex: "XOX------"), Move (INT, 0-8), Result (VARCHAR(10), 'Win', 'Draw', 'Loss'), Frequency (INT, default 1).
  + **Tabela GameResults**:
    - Registra o resultado de cada partida com referência ao estado.
    - Campos: Id (PK), BoardStateId (FK para BoardStates.Id), GameOutcome (VARCHAR(10), 'Win', 'Draw', 'Loss'), CreatedAt (DATETIME).
* **Operações**:
  + Inserir novos estados e resultados após cada jogada.
  + Atualizar Frequency para estados repetidos.
  + Consultar estados por Board e Result para decisões da IA.
  + Garantir integridade referencial entre GameResults e BoardStates.

**2.4. API**

* **Endpoint POST /api/Game/move**:
  + **Descrição**: Processa a jogada do usuário e retorna a jogada da IA.
  + **Entrada**: JSON com o estado do tabuleiro { "Board": "XOX------" }.
  + **Saída**: JSON com o novo estado, jogada da IA, e resultado { "NewBoard": "XOX---O--", "Move": 6, "Result": null | "Win" | "Draw" | "Loss" }.
* **Endpoint GET /api/Game/reset**:
  + **Descrição**: Reinicia o jogo, retornando um tabuleiro vazio.
  + **Saída**: JSON { "Board": "---------" }.
* **Validação**:
  + Verificar se o tabuleiro tem 9 caracteres ('X', 'O', ou '-').
  + Garantir que a jogada do usuário é em uma célula vazia.
  + Retornar HTTP 400 para entradas inválidas.

**3. Requisitos Não Funcionais**

* **Desempenho**:
  + Resposta da IA em menos de 1 segundo por jogada, mesmo com 10.000 registros no banco.
  + Consultas SQL otimizadas com índices para busca rápida.
* **Escalabilidade**:
  + Suportar até 100 usuários simultâneos, com conexão eficiente ao banco.
  + Banco preparado para armazenar até 1 milhão de estados sem degradação significativa.
* **Segurança**:
  + Validar todas as entradas no backend para evitar injeção SQL e XSS.
  + Proteger endpoints contra abuso (ex: limitar taxa de requisições).
* **Usabilidade**:
  + Interface intuitiva, com feedback visual imediato (ex: animação ao clicar).
  + Compatível com navegadores modernos (Chrome, Firefox, Edge, Safari).
  + Layout responsivo para dispositivos desktop e mobile (mínimo 320px de largura).
* **Manutenibilidade**:
  + Código modular, com separação clara de responsabilidades (MVC no backend, componentes no frontend).
  + Comentários no código para funções críticas.
  + Documentação de APIs e instruções de implantação.
* **Confiabilidade**:
  + Garantir consistência entre frontend e backend (ex: tabuleiro sincronizado).
  + Tratar erros graciosamente (ex: falha de conexão ao banco).
* **Portabilidade**:
  + Backend compatível com .NET Core para rodar em Windows, Linux, ou containers (ex: Docker).
  + Frontend servido como estático, hospedável em qualquer servidor web.

**4. Arquitetura**

* **Backend**: ASP.NET Web API (.NET Core 6 ou superior), usando Entity Framework Core para acesso ao banco.
* **Frontend**: AngularJS 1.8.2, servido como arquivo estático, com chamadas HTTP para a API.
* **Banco de Dados**: SQL Server (2019 ou superior), com duas tabelas e índices otimizados.
* **Comunicação**: Protocolo HTTP/REST, com JSON como formato de dados.
* **Camadas**:
  + **Apresentação**: AngularJS gerencia a interface e interações.
  + **Aplicação**: ASP.NET Web API processa lógica de negócio e IA.
  + **Dados**: SQL Server armazena estados e resultados.
* **Diagrama de Arquitetura**:
* [Usuário] --> [Frontend: AngularJS]
* |
* v
* [Backend: ASP.NET Web API] --> [SQL Server: BoardStates, GameResults]

**5. Fluxos Principais**

**5.1. Iniciar Jogo**

1. Usuário acessa a página web (index.html).
2. Frontend carrega AngularJS e exibe tabuleiro vazio (---------).
3. Estado inicial é armazenado no $scope.board.
4. Contador de jogos e resultados (vitórias, empates, derrotas) é inicializado em zero.

**5.2. Jogada do Usuário**

1. Usuário clica em uma célula vazia do tabuleiro.
2. Frontend verifica localmente se a célula está vazia e o jogo não terminou.
3. Marca 'X' na célula e atualiza $scope.board.
4. Envia o tabuleiro ao endpoint /api/Game/move via POST.
5. Exibe um indicador de carregamento (ex: "IA pensando...").

**5.3. Decisão da IA**

1. Backend recebe o tabuleiro e valida a entrada.
2. Identifica posições vazias (onde Board[i] == '-').
3. **Verifica Vitória Imediata**:
   * Simula cada jogada possível para 'O' e verifica se resulta em três 'O' alinhados.
   * Se encontrado, seleciona a jogada.
4. **Verifica Bloqueio**:
   * Simula cada jogada possível para 'X' e verifica se resulta em três 'X' alinhados.
   * Se encontrado, seleciona a jogada para bloquear.
5. **Consulta Banco**:
   * Busca em BoardStates por estados onde Board coincide com o tabuleiro simulado (após jogada 'O') e Result é 'Win' ou 'Draw'.
   * Ordena por Frequency descendente e seleciona a jogada com maior frequência.
6. **Jogada Aleatória**:
   * Se nenhuma jogada foi selecionada, escolhe uma posição vazia aleatoriamente usando Random.
7. **Atualiza Tabuleiro**:
   * Marca 'O' na posição escolhida.
   * Verifica resultado (vitória, empate, ou em andamento).
8. **Salva Estado**:
   * Insere ou atualiza registro em BoardStates com o novo estado, jogada, e resultado.
   * Se o jogo terminou, insere registro em GameResults.
9. Retorna JSON com { NewBoard, Move, Result }.

**5.4. Atualização da Interface**

1. Frontend recebe a resposta da API.
2. Atualiza $scope.board com NewBoard.
3. Incrementa contador de jogos/resultados com base em Result.
4. Exibe mensagem de resultado, se aplicável (ex: "Empate!").
5. Desativa cliques no tabuleiro se o jogo terminou.
6. Remove indicador de carregamento.

**5.5. Reiniciar Jogo**

1. Usuário clica no botão "Reiniciar".
2. Frontend envia requisição GET para /api/Game/reset.
3. Backend retorna tabuleiro vazio (---------).
4. Frontend atualiza $scope.board, limpa resultado, e habilita cliques.

**5.6. Tratamento de Erros**

1. **Erro de Validação**: Backend retorna HTTP 400 com mensagem (ex: "Célula ocupada").
2. **Erro de Conexão**: Frontend exibe alerta (ex: "Falha na conexão. Tente novamente.").
3. **Erro no Banco**: Backend registra log e retorna HTTP 500 com mensagem genérica.

**6. Estrutura do Banco de Dados**

* **Tabela BoardStates**:
* CREATE TABLE BoardStates (
* Id INT IDENTITY(1,1) PRIMARY KEY,
* Board VARCHAR(9) NOT NULL,
* Move INT NOT NULL CHECK (Move BETWEEN 0 AND 8),
* Result VARCHAR(10) NOT NULL CHECK (Result IN ('Win', 'Draw', 'Loss')),
* Frequency INT DEFAULT 1,
* CONSTRAINT CHK\_Board\_Format CHECK (LEN(Board) = 9 AND Board NOT LIKE '%[^X,O,-]%')
* );
* CREATE INDEX IX\_BoardStates\_Board\_Result ON BoardStates(Board, Result);
* **Tabela GameResults**:
* CREATE TABLE GameResults (
* Id INT IDENTITY(1,1) PRIMARY KEY,
* BoardStateId INT NOT NULL FOREIGN KEY REFERENCES BoardStates(Id),
* GameOutcome VARCHAR(10) NOT NULL CHECK (GameOutcome IN ('Win', 'Draw', 'Loss')),
* CreatedAt DATETIME DEFAULT GETDATE()
* );
* **Índices**:
  + Índice em BoardStates(Board, Result) para otimizar consultas da IA.
  + Índice automático em BoardStateId devido à chave estrangeira.

**7. Tecnologias**

* **Backend**:
  + **Linguagem**: C# 10.
  + **Framework**: ASP.NET Core 6 Web API.
  + **ORM**: Entity Framework Core 6.
  + **Pacotes NuGet**:
    - Microsoft.EntityFrameworkCore.SqlServer
    - Microsoft.EntityFrameworkCore.Design (para migrations).
  + **Ferramentas**: Visual Studio 2022 ou Rider.
* **Frontend**:
  + **Framework**: AngularJS 1.8.2 (via CDN: https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js).
  + **Estilização**: CSS puro (grid para tabuleiro, hover effects).
  + **Ferramentas**: VS Code para edição.
* **Banco de Dados**:
  + **SGBD**: SQL Server 2019 ou superior.
  + **Ferramentas**: SQL Server Management Studio (SSMS) para gerenciamento.
* **Servidor**:
  + **Desenvolvimento**: Kestrel (embutido no .NET Core).
  + **Produção**: IIS (Windows) ou Nginx (Linux) para hospedar API e frontend.
* **Outros**:
  + **Testes**: xUnit para testes unitários, Postman para testes de API.
  + **CI/CD**: Opcional (ex: GitHub Actions para build e deploy).
  + **Controle de Versão**: Git (repositório local ou remoto, ex: GitHub).

**8. Estrutura do Projeto**

* **Backend** (TicTacToeApi):
* /TicTacToeApi
* ├── /Controllers
* │ └── GameController.cs
* ├── /Models
* │ ├── BoardState.cs
* │ └── GameResult.cs
* ├── /Data
* │ └── GameContext.cs
* ├── /Properties
* │ └── launchSettings.json
* ├── appsettings.json
* ├── Program.cs
* └── TicTacToeApi.csproj
* **Frontend** (TicTacToeWeb):
* /TicTacToeWeb
* ├── index.html
* └── /assets
* ├── /css
* │ └── styles.css
* └── /js
* └── app.js (opcional, se AngularJS for separado)
* **Banco**:
  + DatabaseSchema.sql: Script para criar tabelas e índices.
  + InitialData.sql (opcional): Dados iniciais para testes.

**9. Implementação Detalhada**

**9.1. Backend**

* **Configuração**:
  + Criar projeto ASP.NET Core Web API (dotnet new webapi).
  + Configurar Entity Framework Core em Program.cs:
  + builder.Services.AddDbContext<GameContext>(options =>
  + options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
  + Definir conexão em appsettings.json:
  + {
  + "ConnectionStrings": {
  + "DefaultConnection": "Server=your\_server;Database=TicTacToeDB;Trusted\_Connection=True;"
  + }
  + }
* **Modelos**:
* public class BoardState
* {
* public int Id { get; set; }
* public string Board { get; set; }
* public int Move { get; set; }
* public string Result { get; set; }
* public int Frequency { get; set; }
* }
* public class GameResult
* {
* public int Id { get; set; }
* public int BoardStateId { get; set; }
* public BoardState BoardState { get; set; }
* public string GameOutcome { get; set; }
* public DateTime CreatedAt { get; set; }
* }
* **Contexto**:
* public class GameContext : DbContext
* {
* public GameContext(DbContextOptions<GameContext> options) : base(options) { }
* public DbSet<BoardState> BoardStates { get; set; }
* public DbSet<GameResult> GameResults { get; set; }
* }
* **Controlador** (GameController.cs):
  + Implementar endpoints /api/Game/move e /api/Game/reset.
  + Lógica de IA com verificação de vitória, bloqueio, consulta ao banco, e jogada aleatória.
  + Exemplo de /move:
  + [HttpPost("move")]
  + public async Task<IActionResult> MakeMove([FromBody] GameMoveRequest request)
  + {
  + if (string.IsNullOrEmpty(request.Board) || request.Board.Length != 9 ||
  + request.Board.Any(c => c != 'X' && c != 'O' && c != '-'))
  + return BadRequest("Invalid board format.");
  + var board = request.Board;
  + var availableMoves = Enumerable.Range(0, 9).Where(i => board[i] == '-').ToList();
  + if (!availableMoves.Any()) return BadRequest("No moves available.");
  + int bestMove = await FindBestMove(board, availableMoves);
  + if (bestMove == -1)
  + bestMove = availableMoves[new Random().Next(availableMoves.Count)];
  + var newBoard = board.ToCharArray();
  + newBoard[bestMove] = 'O';
  + var result = CheckGameResult(newBoard);
  + var boardState = new BoardState
  + {
  + Board = new string(newBoard),
  + Move = bestMove,
  + Result = result ?? "Ongoing",
  + Frequency = 1
  + };
  + var existingState = await \_context.BoardStates
  + .FirstOrDefaultAsync(bs => bs.Board == boardState.Board && bs.Move == bestMove);
  + if (existingState != null)
  + {
  + existingState.Frequency++;
  + }
  + else
  + {
  + \_context.BoardStates.Add(boardState);
  + }
  + if (result != null)
  + {
  + \_context.GameResults.Add(new GameResult
  + {
  + BoardStateId = existingState?.Id ?? boardState.Id,
  + GameOutcome = result
  + });
  + }
  + await \_context.SaveChangesAsync();
  + return Ok(new { NewBoard = new string(newBoard), Result = result, Move = bestMove });
  + }

**9.2. Frontend**

* **HTML** (index.html):
* <!DOCTYPE html>
* <html ng-app="TicTacToeApp">
* <head>
* <title>Jogo da Velha</title>
* <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
* <style>
* .board { display: grid; grid-template-columns: repeat(3, 100px); gap: 5px; }
* .cell { width: 100px; height: 100px; border: 1px solid #000; font-size: 40px; text-align: center; cursor: pointer; }
* .cell.x { color: blue; }
* .cell.o { color: red; }
* .cell:hover:not(.x):not(.o) { background-color: #f0f0f0; }
* .disabled { cursor: not-allowed; }
* .stats { margin-top: 20px; }
* </style>
* </head>
* <body ng-controller="GameController">
* <h1>Jogo da Velha</h1>
* <div class="board">
* <div class="cell" ng-class="{ 'x': cell == 'X', 'o': cell == 'O', 'disabled': cell != '-' || result }"
* ng-repeat="cell in board" ng-click="makeMove($index)" ng-bind="cell"></div>
* </div>
* <p ng-if="result">Resultado: {{ result }}</p>
* <div class="stats">
* <p>Jogos: {{ stats.games }} | Vitórias: {{ stats.wins }} | Empates: {{ stats.draws }} | Derrotas: {{ stats.losses }}</p>
* </div>
* <button ng-click="reset()">Reiniciar</button>
* <p ng-if="error" style="color: red;">{{ error }}</p>
* <script>
* angular.module('TicTacToeApp', [])
* .controller('GameController', ['$scope', '$http', function($scope, $http) {
* $scope.board = ['-', '-', '-', '-', '-', '-', '-', '-', '-'];
* $scope.result = '';
* $scope.error = '';
* $scope.stats = { games: 0, wins: 0, draws: 0, losses: 0 };
* $scope.makeMove = function(index) {
* if ($scope.board[index] !== '-' || $scope.result) return;
* $scope.board[index] = 'X';
* $scope.error = '';
* var boardStr = $scope.board.join('');
* $http.post('/api/Game/move', { Board: boardStr })
* .then(function(response) {
* $scope.board = response.data.NewBoard.split('');
* $scope.result = response.data.Result || '';
* if (response.data.Result) {
* $scope.stats.games++;
* if (response.data.Result === 'Win') $scope.stats.losses++;
* else if (response.data.Result === 'Loss') $scope.stats.wins++;
* else if (response.data.Result === 'Draw') $scope.stats.draws++;
* }
* }, function(error) {
* $scope.error = 'Erro ao processar jogada: ' + (error.data?.message || 'Tente novamente.');
* $scope.board[index] = '-'; // Reverter jogada
* });
* };
* $scope.reset = function() {
* $http.get('/api/Game/reset').then(function(response) {
* $scope.board = response.data.Board.split('');
* $scope.result = '';
* $scope.error = '';
* });
* };
* }]);
* </script>
* </body>
* </html>
* **Estilização**:
  + Usar CSS Grid para o tabuleiro.
  + Estilizar 'X' (azul) e 'O' (vermelho) com cores distintas.
  + Adicionar hover em células vazias e desabilitar cursor em células ocupadas.

**9.3. Banco de Dados**

* Executar script SQL fornecido na seção 6.
* Configurar string de conexão no backend.
* Aplicar migrations iniciais (se usar EF Core):
* dotnet ef migrations add InitialCreate
* dotnet ef database update

**10. Testes**

**10.1. Testes Unitários (Backend)**

* **Ferramenta**: xUnit.
* **Casos**:
  + **Vitória Imediata**: Simular tabuleiro onde 'O' pode vencer (ex: "OO-X-----") e verificar se a IA escolhe a jogada correta.
  + **Bloqueio**: Simular tabuleiro onde 'X' pode vencer (ex: "XX-O-----") e verificar se a IA bloqueia.
  + **Consulta ao Banco**: Mockar GameContext e verificar se a IA escolhe jogada com maior Frequency.
  + **Jogada Aleatória**: Simular tabuleiro sem dados no banco e verificar se a jogada é válida.
  + **Validação**: Testar entradas inválidas (ex: tabuleiro com 8 caracteres) e esperar HTTP 400.
* **Exemplo**:
* [Fact]
* public async Task MakeMove\_WinOpportunity\_ReturnsWinningMove()
* {
* var context = new Mock<GameContext>();
* var controller = new GameController(context.Object);
* var request = new GameMoveRequest { Board = "OO-X-----" };
* var result = await controller.MakeMove(request) as OkObjectResult;
* Assert.NotNull(result);
* var response = result.Value as dynamic;
* Assert.Equal(2, response.Move); // Posição para vitória
* }

**10.2. Testes de Integração**

* **Ferramenta**: Postman ou scripts HTTP.
* **Casos**:
  + Enviar tabuleiro válido para /api/Game/move e verificar resposta (novo tabuleiro, jogada, resultado).
  + Enviar tabuleiro inválido e verificar erro HTTP 400.
  + Simular jogo completo (múltiplas jogadas) e verificar salvamento no banco.
  + Verificar reinício com /api/Game/reset.

**10.3. Testes Manuais**

* Jogar partidas completas para confirmar:
  + Interface atualiza corretamente após jogadas.
  + IA aprende (ex: evita derrotas após repetições do mesmo padrão).
  + Resultados e estatísticas são exibidos corretamente.
  + Reinício funciona sem erros.
* Testar em diferentes navegadores e tamanhos de tela (desktop e mobile).

**10.4. Testes de Carga**

* Simular 100 requisições simultâneas para /api/Game/move usando ferramentas como JMeter.
* Verificar se o tempo de resposta permanece abaixo de 1 segundo.

**11. Implantação**

* **Ambiente de Desenvolvimento**:
  + Rodar backend com dotnet run (Kestrel).
  + Servir frontend com servidor estático (ex: http-server do Node.js ou IIS Express).
* **Ambiente de Produção**:
  + **Backend**:
    - Hospedar em IIS (Windows) ou Nginx (Linux) com reverse proxy para Kestrel.
    - Configurar string de conexão para SQL Server de produção.
    - Habilitar CORS se frontend e backend estiverem em domínios diferentes:
    - builder.Services.AddCors(options =>
    - {
    - options.AddPolicy("AllowAll", builder => builder.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());
    - });
  + **Frontend**:
    - Hospedar arquivos estáticos em servidor web (IIS, Nginx, ou CDN).
    - Configurar URL da API no AngularJS (ex: variável global ou arquivo de configuração).
  + **Banco**:
    - Implantar schema SQL em SQL Server de produção.
    - Configurar backups automáticos e monitoramento.
* **Pipeline CI/CD** (opcional):
  + Usar GitHub Actions para build, testes, e deploy.
  + Exemplo:
  + name: CI/CD
  + on: [push]
  + jobs:
  + build:
  + runs-on: ubuntu-latest
  + steps:
  + - uses: actions/checkout@v3
  + - name: Setup .NET
  + uses: actions/setup-dotnet@v3
  + with: { dotnet-version: '6.0.x' }
  + - name: Build
  + run: dotnet build
  + - name: Test
  + run: dotnet test
  + - name: Deploy
  + run: dotnet publish -o publish

**12. Documentação**

* **Para Desenvolvedores**:
  + **README.md**:
    - Instruções de configuração (pré-requisitos, instalação, build).
    - Como executar testes.
    - Como implantar em desenvolvimento e produção.
  + **Documentação da API**:
    - Descrever endpoints (/move, /reset) com exemplos de entrada/saída.
    - Usar Swagger (opcional) para documentação interativa:
    - builder.Services.AddSwaggerGen();
    - app.UseSwagger();
    - app.UseSwaggerUI();
* **Para Usuários**:
  + **Guia do Usuário**:
    - Explicar como jogar (clicar em células, reiniciar).
    - Descrever feedback visual (cores, mensagens).
    - Mencionar que a IA aprende com o tempo.
  + Incluir no frontend (ex: link "Como Jogar").

**13. Manutenção**

* **Monitoramento**:
  + Configurar logs no backend (ex: Serilog) para erros e consultas lentas.
  + Monitorar uso do banco (ex: tamanho da tabela BoardStates).
* **Atualizações**:
  + Aplicar patches de segurança para .NET, AngularJS, e SQL Server.
  + Revisar desempenho periodicamente e otimizar índices se necessário.
* **Limpeza do Banco**:
  + Implementar script para arquivar ou remover estados com baixa frequência (ex: Frequency < 5 após 6 meses).
  + Exemplo:
  + DELETE FROM BoardStates WHERE Frequency < 5 AND Id NOT IN (SELECT BoardStateId FROM GameResults WHERE CreatedAt > DATEADD(MONTH, -6, GETDATE()));

**14. Extensões Futuras**

* **Autenticação**: Adicionar login para rastrear jogos por usuário.
* **Modos de Jogo**: Permitir jogar contra outro usuário ou IA com níveis de dificuldade.
* **IA Avançada**: Implementar Q-Learning ou redes neurais para aprendizado mais sofisticado.
* **Estatísticas Avançadas**: Mostrar histórico detalhado de jogos e progresso da IA.
* **Multiplataforma**: Criar aplicativos móveis usando .NET MAUI, reutilizando a API.

**15. Premissas**

* Banco SQL Server já está configurado e acessível.
* Ambiente de desenvolvimento inclui Visual Studio 2022, SSMS, e navegador moderno.
* Não há requisitos de autenticação ou suporte a múltiplos usuários simultâneos na mesma sessão.
* Aprendizado da IA é baseado em frequência de padrões, sem algoritmos complexos.

**16. Riscos e Mitigações**

* **Risco**: Banco cresce exponencialmente com estados redundantes.
  + **Mitigação**: Incrementar Frequency para estados existentes e implementar limpeza periódica.
* **Risco**: Resposta lenta da IA com muitos registros.
  + **Mitigação**: Usar índices otimizados e cache para consultas frequentes.
* **Risco**: Bugs na lógica de vitória ou bloqueio.
  + **Mitigação**: Testes unitários abrangentes e validação manual de cenários críticos.
* **Risco**: Interface não responsiva em dispositivos móveis.
  + **Mitigação**: Testar layout em diferentes resoluções e usar CSS media queries.
* **Risco**: Falhas de conexão ao banco.
  + **Mitigação**: Implementar retry policy no EF Core e fallback para jogada aleatória.

**17. Cronograma Detalhado**

* **Semana 1** (Configuração e Planejamento):
  + Configurar projetos (backend, frontend, banco).
  + Criar schema SQL e índices.
  + Definir modelos e contexto do EF Core.
* **Semana 2** (Backend):
  + Implementar GameController com lógica da IA.
  + Configurar endpoints /move e /reset.
  + Integrar EF Core para salvar e consultar estados.
* **Semana 3** (Frontend):
  + Desenvolver index.html com AngularJS.
  + Implementar tabuleiro, jogadas, e estatísticas.
  + Integrar chamadas à API.
* **Semana 4** (Testes e Ajustes):
  + Escrever testes unitários e de integração.
  + Realizar testes manuais em diferentes navegadores.
  + Corrigir bugs e otimizar desempenho.
* **Semana 5** (Implantação e Documentação):
  + Implantar em ambiente de desenvolvimento.
  + Configurar servidor de produção (opcional).
  + Finalizar documentação (README, guia do usuário).
* **Total**: 5 semanas (200 horas, assumindo 1 desenvolvedor em tempo integral).

**18. Entregáveis**

* **Código**:
  + Repositório com backend (TicTacToeApi) e frontend (TicTacToeWeb).
  + Script SQL para banco (DatabaseSchema.sql).
* **Documentação**:
  + README.md com instruções de configuração e execução.
  + Guia do usuário (PDF ou seção no frontend).
  + Documentação da API (Swagger ou Markdown).
* **Testes**:
  + Conjunto de testes unitários (xUnit).
  + Relatório de testes manuais.
* **Implantação**:
  + Backend rodando em servidor local (ex: [http://localhost:5000](http://localhost:5000/)).
  + Frontend acessível via navegador (ex: [http://localhost:8080](http://localhost:8080/)).

**19. Referências**

* **ASP.NET Core**: <https://learn.microsoft.com/en-us/aspnet/core/>
* **Entity Framework Core**: <https://learn.microsoft.com/en-us/ef/core/>
* **AngularJS**: <https://angularjs.org/>
* **SQL Server**: <https://learn.microsoft.com/en-us/sql/>
* **xUnit**: <https://xunit.net/>
* **Swagger**: <https://swagger.io/>

**20. Aprovação**

* **Stakeholders**:
  + Desenvolvedor: Responsável pela implementação.
  + Cliente/Usuário: Valida requisitos e interface final.
* **Critérios de Aceitação**:
  + Usuário pode jogar partidas completas contra a IA.
  + IA aprende com jogos anteriores, evitando derrotas repetidas.
  + Interface é responsiva e exibe resultados/estatísticas corretamente.
  + Banco armazena estados e resultados sem erros.
  + Testes unitários cobrem 80% da lógica crítica.
  + Sistema é implantado e acessível em ambiente de desenvolvimento.

Esta especificação é completa, detalhando todas as fases do desenvolvimento, desde a concepção até a manutenção, com instruções claras para garantir a entrega de um software funcional, robusto e alinhado aos requisitos.