Environment Configuration

```
bash
```

```
# .env file for Replit Secrets
NODE_ENV=development
PORT=3000

# Supabase Database Configuration
SUPABASE_URL="${SUPABASE_URL}"
SUPABASE_ANON_KEY="${SUPABASE_ANON_KEY}"
SUPABASE_SERVICE_KEY="${SUPABASE_SERVICE_KEY}"

# External Redis Configuration
REDIS_URL="${REDIS_URL}" # Redis Cloud, Upstash, or other external provider

# Selectable LLM Configuration
LLM_PROVIDER="writer" # Options: "writer" or "openai"

# Writer API Configuration
WRITER_API_KEY="${WRITER_API_KEY}"
WRITER_BASE_URL="https://api.writer.com/v1"
WRITER_MODEL="palmyra-x-5"

# OpenAI API Configuration
OPENAI_API_KEY="${OPENAI_API_KEY}"
OPENAI_BASE_URL="https://api.openai.com/v1"
OPENAI_MODEL="gpt-4o"

# Vector Database
PINECONE_API_KEY="${PINECONE_API_KEY}"
PINECONE_ENVIRONMENT="${PINECONE_ENVIRONMENT}"
PINECONE_INDEX="${PINECONE_INDEX}"

# External APIs
DATABRICKS_HOST="${DATABRICKS_HOST}"
DATABRICKS_TOKEN="${DATABRICKS_TOKEN}"
DATABRICKS_WAREHOUSE_ID="${DATABRICKS_WAREHOUSE_ID}"

# Security
JWT_SECRET="${JWT_SECRET}"
SESSION_SECRET="${SESSION_SECRET}"

# Feature Flags
ENABLE_REAL_TIME=true
ENABLE_BACKGROUND_JOBS=true
ENABLE_LLM_SWITCHING=true
```

```
LOG_LEVEL=debug

# Cost Management
MAX_TOKENS_PER_REQUEST=8000
DAILY_TOKEN_LIMIT_PER_USER=100000
COST_ALERT_THRESHOLD=100.00
```

LLM Provider Selection Interface

```
typescript
```

```
// Admin interface for LLM provider management
class LLMProviderManager {
  private currentProvider: string;
  private usageTracking: Map<string, ProviderUsage> = new Map();

  constructor() {
    this.currentProvider = process.env.LLM_PROVIDER || 'writer';
  }

  async switchProvider(
    newProvider: 'writer' | 'openai',
    userId?: string
  ): Promise<SwitchResult> {

    // Validate provider availability
    if (!this.isProviderAvailable(newProvider)) {
      throw new Error(`Provider ${newProvider} is not available or configured`);
    }

    // For user-specific switching
    if (userId) {
      await this.setUserProvider(userId, newProvider);
      return {
        success: true,
        provider: newProvider,
        scope: 'user',
        userId
      };
    }

    // Global provider switching (admin only)
    this.currentProvider = newProvider;
    process.env.LLM_PROVIDER = newProvider;

    // Log the switch
    await this.logProviderSwitch(newProvider, 'global');

    return {
      success: true,
      provider: newProvider,
      scope: 'global'
    };
  }
```

```
async getProviderStatus(): Promise<ProviderStatus> {
  const providers = await Promise.all([
    this.checkProviderHealth('writer'),
    this.checkProviderHealth('openai')
  ]);

  return {
    current: this.currentProvider,
    available: providers.filter(p => p.healthy).map(p => p.name),
    status: providers,
    usage: this.getUsageStats()
  };
}

private async checkProviderHealth(provider: string): Promise<ProviderHealth> {
  try {
    const startTime = Date.now();

    if (provider === 'writer') {
      const response = await fetch('https://api.writer.com/v1/health', {
        headers: { 'Authorization': `Bearer ${process.env.WRITER_API_KEY}` }
      });
      return {
        name: provider,
        healthy: response.ok,
        latency: Date.now() - startTime,
        lastChecked: new Date()
      };
    } else if (provider === 'openai') {
      const response = await fetch('https://api.openai.com/v1/models', {
        headers: { 'Authorization': `Bearer ${process.env.OPENAI_API_KEY}` }
      });
      return {
        name: provider,
        healthy: response.ok,
        latency: Date.now() - startTime,
        lastChecked: new Date()
      };
    }

    throw new Error('Unknown provider');
  } catch (error) {
    return {
        name: provider,
        healthy: false,
        error: error.message,
        lastChecked: new Date()
      };
    }
  }

  private async setUserProvider(userId: string, provider: string): Promise<void> {
    await supabase
      .from('users')
      .update({
        preferences: {
          ...await this.getUserPreferences(userId),
          llm_provider: provider
        }
      })
      .eq('id', userId);
  }

  async getUserProvider(userId: string): Promise<string> {
    const { data } = await supabase
      .from('users')
      .select('preferences')
      .eq('id', userId)
      .single();

    return data?.preferences?.llm_provider || this.currentProvider;
  }
}

// Usage tracking and cost management
class LLMUsageTracker {
  async trackUsage(usage: LLMUsageEvent): Promise<void> {
    // Store in Supabase
    await supabase
      .from('llm_usage')
      .insert({
        user_id: usage.userId,
        session_id: usage.sessionId,
        provider: usage.provider,
        model: usage.model,
        tokens_input: usage.tokensInput,
        tokens_output: usage.tokensOutput,
      cost_estimate: this.calculateCost(usage),
      operation_type: usage.operationType
    });

  // Check if user is approaching limits
  await this.checkUsageLimits(usage.userId);
}

private calculateCost(usage: LLMUsageEvent): number {
  const pricing = {
    'writer': {
      'palmyra-x-5': {
        input: 0.003,  // per 1K tokens
        output: 0.015  // per 1K tokens
      }
    },
    'openai': {
      'gpt-4o': {
        input: 0.005,  // per 1K tokens
        output: 0.015  // per 1K tokens
      }
    }
  };

  const modelPricing = pricing[usage.provider]?.[usage.model];
  if (!modelPricing) return 0;

  const inputCost = (usage.tokensInput / 1000) * modelPricing.input;
  const outputCost = (usage.tokensOutput / 1000) * modelPricing.output;

  return inputCost + outputCost;
}

async getDailyUsage(userId: string): Promise<DailyUsage> {
  const today = new Date().toISOString().split('T')[0];

  const { data } = await supabase
    .from('llm_usage')
    .select('*')
    .eq('user_id', userId)
    .gte('created_at', `${today}T00:00:00Z`)
    .lt('created_at', `${today}T23:59:59Z`);

  if (!data) return { tokens: 0, cost: 0, requests: 0 };
```

```
    return {
      tokens: data.reduce((sum, row) => sum + row.tokens_input + row.tokens_output, 0),
      cost: data.reduce((sum, row) => sum + row.cost_estimate, 0),
      requests: data.length,
      byProvider: this.groupByProvider(data)
    };
  }

  private async checkUsageLimits(userId: string): Promise<void> {
    const dailyUsage = await this.getDailyUsage(userId);
    const dailyLimit = parseInt(process.env.DAILY_TOKEN_LIMIT_PER_USER || '100000'); // Doubled
    const costThreshold = parseFloat(process.env.COST_ALERT_THRESHOLD || '100.00');

    // Check token limits
    if (dailyUsage.tokens > dailyLimit * 0.8) {
      await this.sendUsageAlert(userId, 'token_limit_warning', {
        current: dailyUsage.tokens,
        limit: dailyLimit,
        percentage: (dailyUsage.tokens / dailyLimit) * 100
      });
    }

    // Check cost thresholds
    if (dailyUsage.cost > costThreshold * 0.8) {
      await this.sendUsageAlert(userId, 'cost_threshold_warning', {
        current: dailyUsage.cost,
        threshold: costThreshold,
        percentage: (dailyUsage.cost / costThreshold) * 100
      });
    }
  }
}
```

Supabase Real-time Integration

```
typescript
```

```
// Real-time alert system using Supabase
class SupabaseRealtimeService {
  private supabase = createClient(
    process.env.SUPABASE_URL!,
    process.env.SUPABASE_ANON_KEY!
  );

  async setupRealtimeSubscriptions(userId: string): Promise<void> {
    // Subscribe to alerts for the user
    this.supabase
      .channel('user-alerts')
      .on(
        'postgres_changes',
        {
          event: 'INSERT',
          schema: 'public',
          table: 'alerts',
          filter: `user_id=eq.${userId}`
        },
        (payload) => {
          this.handleNewAlert(payload.new as Alert);
        }
      )
      .on(
        'postgres_changes',
        {
          event: 'INSERT',
          schema: 'public',
          table: 'competitive_events',
          filter: `revenue_impact=gte.20000` // Only high-impact events
        },
        (payload) => {
          this.handleCompetitiveEvent(payload.new as CompetitiveEvent);
        }
      )
      .subscribe();

    // Subscribe to conversation updates for active sessions
    this.supabase
      .channel('conversations')
      .on(
        'postgres_changes',
        {
```

```
        event: 'INSERT',
        schema: 'public',
        table: 'conversations'
      },
      (payload) => {
        this.handleConversationUpdate(payload.new as Conversation);
      }
    )
    .subscribe();
}

private async handleNewAlert(alert: Alert): Promise<void> {
  // Emit to connected WebSocket clients
  this.emitToUser(alert.user_id, 'new-alert', alert);

  // Send push notification if critical
  if (alert.priority === 'critical') {
    await this.sendPushNotification(alert.user_id, {
      title: alert.title,
      body: alert.message,
      data: { alertId: alert.id, type: 'critical-alert' }
    });
  }
}

private async handleCompetitiveEvent(event: CompetitiveEvent): Promise<void> {
  // Find affected users based on route portfolio
  const { data: affectedUsers } = await supabase
    .from('users')
    .select('id, route_portfolio')
    .contains('route_portfolio', [event.route]);

  if (affectedUsers) {
    for (const user of affectedUsers) {
      this.emitToUser(user.id, 'competitive-event', event);
    }
  }
}

async broadcastSystemMessage(message: SystemMessage): Promise<void> {
  // Broadcast to all connected users
  this.supabase
    .channel('system-broadcasts')
    .send({
      type: 'broadcast',
      event: 'system-message',
      payload: message
    });
  }
}

// Supabase database operations with connection pooling
class SupabaseService {
  private supabase = createClient(
    process.env.SUPABASE_URL!,
    process.env.SUPABASE_SERVICE_KEY!
  );

  async createIntelligenceSession(
    userId: string,
    sessionType: SessionType,
    llmProvider?: string
  ): Promise<string> {

    const { data, error } = await this.supabase
      .from('intelligence_sessions')
      .insert({
        user_id: userId,
        session_type: sessionType,
        llm_provider: llmProvider || await this.getUserProvider(userId),
        context: {},
        metadata: {
          ip_address: this.getClientIP(),
          user_agent: this.getUserAgent()
        }
      })
      .select('id')
      .single();

    if (error) throw new DatabaseError(`Failed to create session: ${error.message}`);
    return data.id;
  }

  async storeAgentResult(
    sessionId: string,
    agentType: string,
    result: AgentResult
  ): Promise<void> {
    const { error } = await this.supabase
      .from('agent_results')
      .insert({
        session_id: sessionId,
        agent_type: agentType,
        llm_provider: result.metadata.llmProvider,
        confidence: result.confidence,
        processing_time: result.metadata.processingTime,
        token_usage: result.metadata.tokenUsage || 0,
        insights: result.insights,
        recommendations: result.recommendations,
        raw_data: result.data
      });

    if (error) {
      throw new DatabaseError(`Failed to store agent result: ${error.message}`);
    }
  }

  async createAlert(alert: CreateAlertRequest): Promise<string> {
    const { data, error } = await this.supabase
      .from('alerts')
      .insert({
        user_id: alert.userId,
        alert_type: alert.type,
        priority: alert.priority,
        title: alert.title,
        message: alert.message,
        revenue_impact: alert.revenueImpact,
        route: alert.route,
        source_data: alert.sourceData,
        expires_at: alert.expiresAt
      })
      .select('id')
      .single();

    if (error) throw new DatabaseError(`Failed to create alert: ${error.message}`);
    return data.id;
  }

  async getUserDashboardData(userId: string): Promise<DashboardData> {
    // Use Supabase RPC for complex queries
    const { data, error } = await this.supabase
```

```
      .rpc('get_user_dashboard_data', {
        p_user_id: userId,
        p_date_range: 7 // Last 7 days
      });

    if (error) {
      throw new DatabaseError(`Failed to get dashboard data: ${error.message}`);
    }

    return data;
  }

  private async getUserProvider(userId: string): Promise<string> {
    const { data } = await this.supabase
      .from('users')
      .select('preferences')
      .eq('id', userId)
      .single();

    return data?.preferences?.llm_provider ||
           process.env.LLM_PROVIDER ||
           'writer';
  }
}
```

Development Setup with External Services

```bash
```

```
# Getting Started with External Dependencies

# 1. Set up Supabase Project
# - Create new project at https://supabase.com
# - Copy Project URL and API keys to Replit Secrets
# - Run schema migration in Supabase SQL Editor

# 2. Set up External Redis
# Option A: Redis Cloud (recommended)
# - Create account at https://redis.com/redis-enterprise-cloud/
# Option B: Upstash (serverless Redis)
# - Create account at https://upstash.com

# 3. Configure LLM Providers
# Writer API:
# - Sign up at https://writer.com
# - Generate API key for Palmyra X5 access
# OpenAI API:
# - Create account at https://openai.com
# - Generate API key for GPT-4o access

# 4. Install dependencies in Replit
npm install @supabase/supabase-js ioredis

# 5. Run database migrations
npm run db:migrate

# 6. Seed with sample data
npm run db:seed

# 7. Start development server
npm run dev
```

This updated architecture now properly supports:

✅ **Selectable LLM Providers**: Dynamic switching between Writer Palmyra X5 and OpenAI GPT-4o ✅ **External Supabase Database**: Complete configuration with RLS, real-time subscriptions, and optimized schema ✅ **External Redis**: Support for Redis Cloud, Upstash, or other external Redis providers ✅ **Cost Management**: Token usage tracking and cost estimation for both LLM providers ✅ **Real-time Features**: Supabase real-time subscriptions for alerts and competitive events ✅ **Security**: Row Level Security policies and proper authentication flows

The system is now ready for enterprise deployment with external, scalable infrastructure while maintaining rapid development capabilities in Replit.### **Environment Configuration**

```
bash
```

~~~
# .env file for Replit Secrets
NODE_ENV=development
PORT=3000

# Supabase Database Configuration
SUPABASE_URL="${SUPABASE_URL}"
SUPABASE_ANON_KEY="${SUPABASE_# Velociti Technical Requirements Document
## Replit Implementation Architecture

**Document Version**: 1.0
**Target Platform**: Replit (Node.js/React)
**Development Phase**: MVP Implementation
**Architecture Style**: Microservices with Event-Driven Intelligence

---

## System Overview

**Core Architecture Principle**: Velociti is built as a **distributed intelligence platform** where specialized AI ager

**Replit Implementation Strategy**: Monorepo structure with microservice separation, utilizing Replit's database,

---

## Technical Stack

### **Frontend Stack**
```javascript
// Primary Technologies
React 18+ (with Hooks and Context API)
TypeScript (strict mode)
Tailwind CSS (for styling)
Recharts (for data visualizations)
Socket.io-client (real-time updates)
React Query (data fetching and caching)

// State Management
Zustand (lightweight state management)
React Hook Form (form handling)
~~~

Backend Stack

```javascript
// Core Backend
Node.js 18+ with Express.js
TypeScript (strict mode)
Socket.io (real-time communication)
Prisma ORM (database operations)
Bull Queue (job processing)

// AI Integration
OpenAI SDK (GPT-4 for prototyping, Writer API for production)
LangChain.js (agent orchestration)
Pinecone (vector database for memory)
```

Database & Infrastructure

```javascript
// External Database (Supabase)
Supabase PostgreSQL (external hosted)
Supabase Auth (authentication service)
Supabase Realtime (real-time subscriptions)
Redis (external - Redis Cloud or Upstash)

// AI Provider Selection
Writer Palmyra X5 API (production recommendation)
OpenAI GPT-4o API (alternative/development)
Configurable LLM switching via environment variables
```

Replit-Specific Configurations

```bash
```

```
# .replit configuration
run = "npm run dev"
modules = ["nodejs-18", "web"]

[env]
NODE_ENV = "development"
# External Supabase Database
SUPABASE_URL = "${SUPABASE_URL}"
SUPABASE_ANON_KEY = "${SUPABASE_ANON_KEY}"
SUPABASE_SERVICE_KEY = "${SUPABASE_SERVICE_KEY}"
# External Redis
REDIS_URL = "${REDIS_URL}"
# Selectable LLM Configuration
LLM_PROVIDER = "writer" # or "openai"

[deployment]
deploymentTarget = "cloudrun"
```

Conceptual Architecture

High-Level System Architecture

```
┌──────────────────────────────────────────────────────────────
│                    VELOCITI PLATFORM                          │
├──────────────────────────────────────────────────────────────
│ Frontend Layer (React + TypeScript)                          │
│  ├── Analyst Workbench (Primary Interface)                   │
│  ├── Real-Time Dashboard (Live Updates)                      │
│  ├── Conversational Interface (Chat UI)                      │
│  └── Executive Summary (Mobile-Optimized)                    │
├──────────────────────────────────────────────────────────────
│ API Gateway Layer (Express.js)                               │
│  ├── Authentication & Authorization                          │
│  ├── Rate Limiting & Security                                │
│  ├── Request Routing & Load Balancing                        │
│  └── Real-Time Event Broadcasting                            │
├──────────────────────────────────────────────────────────────
│ Intelligence Orchestration Layer                             │
│  ├── Agent Coordinator (Master Controller)                   │
│  ├── Context Manager (Cross-Agent Memory)                    │
│  ├── Event Bus (Agent Communication)                         │
│  └── Priority Queue (Task Scheduling)                        │
├──────────────────────────────────────────────────────────────
│ Specialized AI Agents                                        │
│  ├── NightShift Agent (Overnight Processing)                 │
│  ├── Competitive Intelligence Agent                          │
│  ├── Performance Attribution Agent                           │
│  ├── Alert Classification Agent                              │
│  └── Data Interrogation Agent (Genie Proxy)                  │
├──────────────────────────────────────────────────────────────
│ Data Integration Layer                                       │
│  ├── Databricks Connector (Primary Data Source)              │
│  ├── Writer API Connector (Strategic AI)                     │
│  ├── Simulated EasyJet Data (Development)                    │
│  └── External APIs (Infare, OAG, Weather)                    │
### **Storage & Memory Layer**
```

```
├──────────────────────────────────────────────────────────────
──────────────────────────────┤
│ Storage & Memory Layer                                       │
│  ├── Supabase PostgreSQL (External Primary Database)         │
│  ├── Supabase Auth (User Authentication Service)             │
│  ├── Supabase Realtime (Real-time Data Subscriptions)        │
```

```
|   ├──── Redis Cloud/Upstash (External Caching & Job Queues)      |
|   ├──── Pinecone (Vector Memory Store)                           |
|   └──── File Storage (Reports, Exports)                          |
 └──────────────────────────────────────────────────────────────────
───────────────────────────────────┘
```

~~~
### **Agent Architecture Design**
```javascript
// Base Agent Interface
interface IntelligenceAgent {
  id: string;
  name: string;
  specialization: string;
  process(context: AgentContext): Promise<AgentResult>;
  subscribe(events: string[]): void;
  publish(event: AgentEvent): void;
}

// Agent Context Structure
interface AgentContext {
  userId: string;
  sessionId: string;
  query?: string;
  data: any;
  previousResults?: AgentResult[];
  timestamp: Date;
  priority: 'critical' | 'high' | 'medium' | 'low';
}

// Agent Result Structure
interface AgentResult {
  agentId: string;
  confidence: number;
  insights: Insight[];
  recommendations: Recommendation[];
  data: any;
  metadata: {
    processingTime: number;
    dataSourcesUsed: string[];
    nextActions?: string[];
  };
}
~~~

Data Flow Architecture

Primary Data Flows

Flow 1: Morning Intelligence Briefing

mermaid

```
graph TD
    A[Scheduled Job 6AM GMT] --> B[NightShift Agent Activation]
    B --> C[Data Collection Phase]
    C --> D[Multi-Agent Processing]
    D --> E[Insight Synthesis]
    E --> F[Priority Ranking]
    F --> G[Briefing Generation]
    G --> H[User Notification]

    C --> C1[Databricks API Call]
    C --> C2[Competitive Data Fetch]
    C --> C3[Performance Metrics Load]

    D --> D1[Competitive Agent Analysis]
    D --> D2[Performance Agent Analysis]
    D --> D3[Alert Classification]

    E --> E1[Cross-Agent Correlation]
    E --> E2[Context Enrichment]
    E --> E3[Confidence Scoring]
```

Flow 2: Real-Time Competitive Intelligence

mermaid

```
graph TD
    A[External Data Stream] --> B[Change Detection]
    B --> C[Impact Assessment]
    C --> D[Alert Classification]
    D --> E[Agent Routing]
    E --> F[Intelligence Processing]
    F --> G[Real-Time Notification]

    B --> B1[Price Change Detection]
    B --> B2[Capacity Change Detection]
    B --> B3[Schedule Change Detection]

    D --> D1[Critical >£50K]
    D --> D2[High £20K-£50K]
    D --> D3[Medium £5K-£20K]
    D --> D4[Low <£5K]

    G --> G1[WebSocket Push]
    G --> G2[Mobile Notification]
    G --> G3[Email Alert]
```

Flow 3: Conversational Intelligence

```
mermaid
```

```
graph TD
    A[User Query] --> B[Intent Classification]
    B --> C[Route Decision]
    C --> D1[Data Query Path - Genie]
    C --> D2[Strategic Analysis Path - Writer]

    D1 --> E1[Databricks Query]
    E1 --> F1[Data Processing]
    F1 --> G1[Visualization Generation]

    D2 --> E2[Context Enrichment]
    E2 --> F2[Writer API Call]
    F2 --> G2[Strategic Recommendation]

    G1 --> H[Response Assembly]
    G2 --> H
    H --> I[Context Storage]
    I --> J[User Response]
```

Data Integration Patterns

Databricks Integration Flow

```
javascript
```

```typescript
// Databricks Connector Implementation
class DatabricksConnector {
 private baseUrl: string;
 private token: string;
 private rateLimiter: RateLimiter;

 async executeQuery(sql: string, context: QueryContext): Promise<QueryResult> {
  // Rate limiting
  await this.rateLimiter.acquire();

  try {
   // Execute query via Databricks SQL API
   const response = await fetch(`${this.baseUrl}/api/2.0/sql/statements`, {
    method: 'POST',
    headers: {
     'Authorization': `Bearer ${this.token}`,
     'Content-Type': 'application/json'
    },
    body: JSON.stringify({
     warehouse_id: process.env.DATABRICKS_WAREHOUSE_ID,
     statement: sql,
     wait_timeout: '30s'
    })
   });

   const result = await response.json();

   // Transform to standard format
   return this.transformResult(result, context);
  } catch (error) {
   // Error handling and fallback
   throw new DatabricksError(`Query failed: ${error.message}`);
  }
 }

 private transformResult(raw: any, context: QueryContext): QueryResult {
  // Transform Databricks response to Velociti format
  return {
   data: raw.result?.data_array || [],
   schema: raw.result?.manifest?.schema?.columns || [],
   executionTime: raw.result?.duration || 0,
   source: 'databricks',
   context: context
```

```
    };
  }
}
```

Selectable LLM Architecture

```
javascript
```

```typescript
// LLM Provider Abstraction Layer
interface LLMProvider {
  name: string;
  model: string;
  generateResponse(prompt: string, context: LLMContext): Promise<LLMResponse>;
  generateEmbedding(text: string): Promise<number[]>;
  estimateTokens(text: string): number;
  maxTokens: number;
}

// LLM Provider Factory
class LLMProviderFactory {
  private static providers: Map<string, LLMProvider> = new Map();

  static registerProvider(name: string, provider: LLMProvider): void {
    this.providers.set(name, provider);
  }

  static getProvider(name?: string): LLMProvider {
    const providerName = name || process.env.LLM_PROVIDER || 'writer';
    const provider = this.providers.get(providerName);

    if (!provider) {
      throw new Error(`LLM provider '${providerName}' not found`);
    }

    return provider;
  }

  static listAvailableProviders(): string[] {
    return Array.from(this.providers.keys());
  }
}

// Writer Palmyra X5 Implementation
class WriterProvider implements LLMProvider {
  name = 'writer';
  model = 'palmyra-x-5';
  maxTokens = 128000;
  private apiKey: string;
  private baseUrl = 'https://api.writer.com/v1';

  constructor() {
    this.apiKey = process.env.WRITER_API_KEY!;
  }

  async generateResponse(prompt: string, context: LLMContext): Promise<LLMResponse> {
    const response = await fetch(`${this.baseUrl}/chat/completions`, {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${this.apiKey}`,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        model: this.model,
        messages: this.buildMessages(prompt, context),
        max_tokens: context.maxTokens || 4000, // Doubled default
        temperature: context.temperature || 0.3,
        stream: false
      })
    });

    if (!response.ok) {
      throw new LLMError(`Writer API error: ${response.statusText}`);
    }

    const result = await response.json();
    return this.transformResponse(result);
  }

  async generateEmbedding(text: string): Promise<number[]> {
    const response = await fetch(`${this.baseUrl}/embeddings`, {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${this.apiKey}`,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        input: text,
        model: 'text-embedding-ada-002' // Writer's embedding model
      })
    });

    const result = await response.json();
    return result.data[0].embedding;
  }
  private buildMessages(prompt: string, context: LLMContext): any[] {
    const messages = [{
      role: 'system',
      content: this.getSystemPrompt(context)
    }];

    // Add conversation history if available
    if (context.conversationHistory) {
      messages.push(...context.conversationHistory);
    }

    messages.push({
      role: 'user',
      content: prompt
    });

    return messages;
  }

  private getSystemPrompt(context: LLMContext): string {
    return `You are Velociti, an AI assistant specialized in airline revenue management for EasyJet.
    You provide strategic analysis with confidence scores and actionable recommendations.

    Context Type: ${context.type}
    User Role: ${context.userRole}

    Always provide:
    1. Strategic analysis with business context
    2. Confidence score (0-100)
    3. Specific recommendations with implementation steps
    4. Risk assessment when applicable`;
  }
}

// OpenAI GPT-4o Implementation
class OpenAIProvider implements LLMProvider {
  name = 'openai';
  model = 'gpt-4o';
  maxTokens = 128000;
  private apiKey: string;
  private baseUrl = 'https://api.openai.com/v1';

  constructor() {
    this.apiKey = process.env.OPENAI_API_KEY!;
```

```
}

async generateResponse(prompt: string, context: LLMContext): Promise<LLMResponse> {
  const response = await fetch(`${this.baseUrl}/chat/completions`, {
    method: 'POST',
    headers: {
      'Authorization': `Bearer ${this.apiKey}`,
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      model: this.model,
      messages: this.buildMessages(prompt, context),
      max_tokens: context.maxTokens || 4000, // Doubled default
      temperature: context.temperature || 0.3,
      response_format: { type: "json_object" } // For structured responses
    })
  });

  if (!response.ok) {
    throw new LLMError(`OpenAI API error: ${response.statusText}`);
  }

  const result = await response.json();
  return this.transformResponse(result);
}

async generateEmbedding(text: string): Promise<number[]> {
  const response = await fetch(`${this.baseUrl}/embeddings`, {
    method: 'POST',
    headers: {
      'Authorization': `Bearer ${this.apiKey}`,
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      input: text,
      model: 'text-embedding-3-large'
    })
  });

  const result = await response.json();
  return result.data[0].embedding;
}

private buildMessages(prompt: string, context: LLMContext): any[] {
    const messages = [{
      role: 'system',
      content: this.getSystemPrompt(context)
    }];

    if (context.conversationHistory) {
      messages.push(...context.conversationHistory);
    }

    messages.push({
      role: 'user',
      content: prompt
    });

    return messages;
  }

  private getSystemPrompt(context: LLMContext): string {
    return `You are Velociti, an AI assistant specialized in airline revenue management for EasyJet.
    Respond in JSON format with the following structure:
    {
      "analysis": "Strategic analysis with business context",
      "confidence": 85,
      "recommendations": [
        {
          "action": "Specific action to take",
          "rationale": "Why this action is recommended",
          "implementation": "How to implement this action",
          "timeline": "When to implement",
          "risk_level": "low|medium|high"
        }
      ],
      "risks": ["List of potential risks"],
      "next_actions": ["Suggested follow-up actions"]
    }

    Context Type: ${context.type}
    User Role: ${context.userRole}`;
  }
}

// Provider Registration and Usage
class LLMService {
  private static instance: LLMService;
private currentProvider: LLMProvider;

private constructor() {
  // Register available providers
  LLMProviderFactory.registerProvider('writer', new WriterProvider());
  LLMProviderFactory.registerProvider('openai', new OpenAIProvider());

  // Set default provider
  this.currentProvider = LLMProviderFactory.getProvider();
}

static getInstance(): LLMService {
  if (!this.instance) {
    this.instance = new LLMService();
  }
  return this.instance;
}

async switchProvider(providerName: string): Promise<void> {
  this.currentProvider = LLMProviderFactory.getProvider(providerName);
  console.log(`Switched to LLM provider: ${providerName}`);
}

getCurrentProvider(): string {
  return this.currentProvider.name;
}

async generateStrategicAnalysis(
  query: string,
  context: AnalysisContext
): Promise<StrategicResponse> {

  const llmContext: LLMContext = {
    type: 'strategic-analysis',
    userRole: context.userRole,
    conversationHistory: context.conversationHistory,
    maxTokens: 4000, // Doubled from 2000
    temperature: 0.3
  };

  const prompt = this.buildAnalysisPrompt(query, context);
  const response = await this.currentProvider.generateResponse(prompt, llmContext);

  return this.parseStrategicResponse(response);
  }

  async generateDataInsight(
    query: string,
    data: any,
    context: DataContext
  ): Promise<DataInsightResponse> {

    const llmContext: LLMContext = {
      type: 'data-insight',
      userRole: context.userRole,
      maxTokens: 3000, // Doubled from 1500
      temperature: 0.1 // Lower temperature for data analysis
    };

    const prompt = this.buildDataInsightPrompt(query, data, context);
    const response = await this.currentProvider.generateResponse(prompt, llmContext);

    return this.parseDataInsightResponse(response);
  }
}
```

~~~~
---

## Real-Time Architecture

### **WebSocket Event System**
```javascript
// Real-Time Event Architecture
interface VelocitiEvent {
  type: 'alert' | 'update' | 'insight' | 'recommendation';
  priority: 'critical' | 'high' | 'medium' | 'low';
  userId: string;
  data: any;
  timestamp: Date;
  source: string;
}

class EventBroadcaster {
  private io: Server;
  private redis: Redis;

  constructor(server: any) {
    this.io = new Server(server, {
      cors: { origin: "*" },
      transports: ['websocket', 'polling']
    });

    this.setupEventHandlers();
  }

  async broadcastToUser(userId: string, event: VelocitiEvent) {
    // Send to connected user sessions
    this.io.to(`user-${userId}`).emit('intelligence-update', event);

    // Store in Redis for offline users
    await this.redis.lpush(`events:${userId}`, JSON.stringify(event));
    await this.redis.expire(`events:${userId}`, 86400); // 24 hours
  }

  async broadcastAlert(alert: AlertEvent) {
    const affectedUsers = await this.getAffectedUsers(alert);

    for (const userId of affectedUsers) {
~~~~

```
      await this.broadcastToUser(userId, {
        type: 'alert',
        priority: alert.priority,
        userId: userId,
        data: alert,
        timestamp: new Date(),
        source: 'competitive-intelligence'
      });
    }
  }
}
```

Job Queue Architecture

```
javascript
```

```
// Background Job Processing
import Bull from 'bull';

class IntelligenceJobQueue {
  private nightShiftQueue: Bull.Queue;
  private alertQueue: Bull.Queue;
  private analysisQueue: Bull.Queue;

  constructor() {
    const redisConfig = { redis: { port: 6379, host: 'localhost' } };

    this.nightShiftQueue = new Bull('night-shift', redisConfig);
    this.alertQueue = new Bull('alerts', redisConfig);
    this.analysisQueue = new Bull('analysis', redisConfig);

    this.setupProcessors();
  }

  private setupProcessors() {
    // Night Shift Processing (Daily at 6 AM GMT)
    this.nightShiftQueue.process('morning-briefing', async (job) => {
      const { userId, preferences } = job.data;

      try {
        // Coordinate multiple agents
        const agents = await this.getActiveAgents();
        const results = await Promise.all(
          agents.map(agent => agent.process({
            userId,
            sessionId: `briefing-${Date.now()}`,
            data: await this.getBaselineData(userId),
            timestamp: new Date(),
            priority: 'high'
          }))
        );

        // Synthesize results
        const briefing = await this.synthesizeBriefing(results, preferences);

        // Store and notify
        await this.storeBriefing(userId, briefing);
        await this.notifyUser(userId, briefing);
      return { success: true, briefingId: briefing.id };
    } catch (error) {
      throw new Error(`Morning briefing failed: ${error.message}`);
    }
  });

  // Real-Time Alert Processing
  this.alertQueue.process('competitive-alert', async (job) => {
    const { changeEvent } = job.data;

    // Classify alert priority
    const classification = await this.classifyAlert(changeEvent);

    if (classification.priority === 'critical') {
      // Immediate processing for critical alerts
      const analysis = await this.getCompetitiveAnalysis(changeEvent);
      await this.broadcastCriticalAlert(analysis);
    }

    return { processed: true, priority: classification.priority };
  });
}

// Schedule recurring jobs
async scheduleRecurringJobs() {
  // Daily morning briefings at 6 AM GMT
  await this.nightShiftQueue.add('morning-briefing',
    { type: 'all-users' },
    {
      repeat: { cron: '0 6 * * *', tz: 'GMT' },
      removeOnComplete: 10,
      removeOnFail: 5
    }
  );

  // Hourly competitive monitoring during business hours
  await this.alertQueue.add('competitive-monitor',
    { type: 'scan-competitors' },
    {
      repeat: { cron: '0 7-19 * * *', tz: 'GMT' }, // 7 AM to 7 PM GMT
      removeOnComplete: 24,
      removeOnFail: 3
    }
  );
  }
}
```

Database Schema Design (Supabase PostgreSQL)

Supabase Configuration

```
javascript
```

```
// Supabase Client Setup
import { createClient } from '@supabase/supabase-js';

const supabaseUrl = process.env.SUPABASE_URL!;
const supabaseKey = process.env.SUPABASE_ANON_KEY!;
const supabaseServiceKey = process.env.SUPABASE_SERVICE_KEY!;

// Client for frontend operations
export const supabase = createClient(supabaseUrl, supabaseKey, {
  auth: {
    persistSession: true,
    detectSessionInUrl: true
  },
  realtime: {
    params: {
      eventsPerSecond: 10
    }
  }
});

// Admin client for backend operations
export const supabaseAdmin = createClient(supabaseUrl, supabaseServiceKey, {
  auth: {
    autoRefreshToken: false,
    persistSession: false
  }
});

// Database types generation
export type Database = {
  public: {
    Tables: {
      users: {
        Row: User;
        Insert: Omit<User, 'id' | 'created_at' | 'updated_at'>;
        Update: Partial<Omit<User, 'id'>>;
      };
      intelligence_sessions: {
        Row: IntelligenceSession;
        Insert: Omit<IntelligenceSession, 'id' | 'started_at'>;
        Update: Partial<Omit<IntelligenceSession, 'id'>>;
      };
      // ... other table types
    };
  };
};
```

Core Data Models (Supabase Schema)

sql

```sql
-- Enable required extensions
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "pg_cron";

-- Enable Row Level Security
ALTER DATABASE postgres SET "app.settings.jwt_secret" TO 'your-jwt-secret';

-- Users table (extends Supabase auth.users)
CREATE TABLE public.users (
  id UUID REFERENCES auth.users(id) ON DELETE CASCADE PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  name VARCHAR(255) NOT NULL,
  role VARCHAR(50) NOT NULL CHECK (role IN ('analyst', 'manager', 'executive')),
  preferences JSONB DEFAULT '{}',
  route_portfolio TEXT[], -- Array of routes user is responsible for
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Enable RLS on users table
ALTER TABLE public.users ENABLE ROW LEVEL SECURITY;

-- Policy: Users can only see their own data
CREATE POLICY "Users can view own profile" ON public.users
  FOR SELECT USING (auth.uid() = id);

CREATE POLICY "Users can update own profile" ON public.users
  FOR UPDATE USING (auth.uid() = id);

-- Intelligence Sessions
CREATE TABLE public.intelligence_sessions (
  id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  user_id UUID REFERENCES public.users(id) ON DELETE CASCADE,
  session_type VARCHAR(50) NOT NULL CHECK (session_type IN ('briefing', 'conversation', 'analysis')),
  llm_provider VARCHAR(20) NOT NULL DEFAULT 'writer' CHECK (llm_provider IN ('writer', 'openai')),
  started_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
  ended_at TIMESTAMP WITH TIME ZONE,
  context JSONB DEFAULT '{}',
  metadata JSONB DEFAULT '{}',
  total_tokens INTEGER DEFAULT 0,
  cost_estimate DECIMAL(10,4) DEFAULT 0
);
```

```sql
ALTER TABLE public.intelligence_sessions ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view own sessions" ON public.intelligence_sessions
  FOR SELECT USING (user_id = auth.uid());

-- Agent Processing Results
CREATE TABLE public.agent_results (
  id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  session_id UUID REFERENCES public.intelligence_sessions(id) ON DELETE CASCADE,
  agent_type VARCHAR(100) NOT NULL,
  llm_provider VARCHAR(20) NOT NULL,
  confidence DECIMAL(5,2) NOT NULL CHECK (confidence >= 0 AND confidence <= 100),
  processing_time INTEGER NOT NULL, -- milliseconds
  token_usage INTEGER DEFAULT 0,
  insights JSONB NOT NULL DEFAULT '[]',
  recommendations JSONB NOT NULL DEFAULT '[]',
  raw_data JSONB,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

ALTER TABLE public.agent_results ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view results from their sessions" ON public.agent_results
  FOR SELECT USING (
    session_id IN (
      SELECT id FROM public.intelligence_sessions WHERE user_id = auth.uid()
    )
  );

-- Alerts and Notifications with Real-time Support
CREATE TABLE public.alerts (
  id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  user_id UUID REFERENCES public.users(id) ON DELETE CASCADE,
  alert_type VARCHAR(100) NOT NULL,
  priority VARCHAR(20) NOT NULL CHECK (priority IN ('critical', 'high', 'medium', 'low')),
  title VARCHAR(500) NOT NULL,
  message TEXT NOT NULL,
  revenue_impact DECIMAL(12,2),
  route VARCHAR(10), -- e.g., 'LGW-BCN'
  source_data JSONB,
  acknowledged_at TIMESTAMP WITH TIME ZONE,
  acknowledged_by UUID REFERENCES public.users(id),
  expires_at TIMESTAMP WITH TIME ZONE,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
```

```
);

ALTER TABLE public.alerts ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view relevant alerts" ON public.alerts
  FOR SELECT USING (
    user_id = auth.uid() OR
    route = ANY(
      SELECT unnest(route_portfolio)
      FROM public.users
      WHERE id = auth.uid()
    )
  );

-- Conversational History
CREATE TABLE public.conversations (
  id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  session_id UUID REFERENCES public.intelligence_sessions(id) ON DELETE CASCADE,
  message_type VARCHAR(50) NOT NULL CHECK (message_type IN ('user_query', 'genie_response', 'writer_resp
  content TEXT NOT NULL,
  llm_provider VARCHAR(20),
  token_count INTEGER DEFAULT 0,
  metadata JSONB DEFAULT '{}',
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

ALTER TABLE public.conversations ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can view conversation history from their sessions" ON public.conversations
  FOR SELECT USING (
    session_id IN (
      SELECT id FROM public.intelligence_sessions WHERE user_id = auth.uid()
    )
  );

-- Performance Metrics (Large table - partitioned)
CREATE TABLE public.performance_metrics (
  id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  metric_date DATE NOT NULL,
  metric_type VARCHAR(100) NOT NULL,
  route VARCHAR(10) NOT NULL, -- e.g., 'LGW-BCN'
  competitor VARCHAR(50), -- 'EasyJet', 'Ryanair', 'Wizz Air', etc.
  value DECIMAL(15,4) NOT NULL,
  baseline DECIMAL(15,4),
  variance_pct DECIMAL(5,2),
  data_source VARCHAR(100) NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
) PARTITION BY RANGE (metric_date);

-- Create partitions for performance metrics
CREATE TABLE public.performance_metrics_2024 PARTITION OF public.performance_metrics
FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

CREATE TABLE public.performance_metrics_2025 PARTITION OF public.performance_metrics
FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');

-- Competitive Intelligence Events
CREATE TABLE public.competitive_events (
  id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  competitor VARCHAR(50) NOT NULL,
  event_type VARCHAR(100) NOT NULL,
  route VARCHAR(10) NOT NULL,
  old_value DECIMAL(10,2),
  new_value DECIMAL(10,2),
  change_pct DECIMAL(5,2),
  detected_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
  processed_at TIMESTAMP WITH TIME ZONE,
  revenue_impact DECIMAL(12,2),
  alert_sent BOOLEAN DEFAULT FALSE,
  data_source VARCHAR(100) NOT NULL
);

-- LLM Usage Tracking
CREATE TABLE public.llm_usage (
  id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  user_id UUID REFERENCES public.users(id),
  session_id UUID REFERENCES public.intelligence_sessions(id),
  provider VARCHAR(20) NOT NULL,
  model VARCHAR(50) NOT NULL,
  tokens_input INTEGER NOT NULL,
  tokens_output INTEGER NOT NULL,
  cost_estimate DECIMAL(10,6) NOT NULL,
  operation_type VARCHAR(50) NOT NULL, -- 'analysis', 'embedding', 'chat', etc.
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Create optimized indexes
CREATE INDEX idx_users_email ON public.users(email);
CREATE INDEX idx_sessions_user_date ON public.intelligence_sessions(user_id, started_at DESC);
CREATE INDEX idx_alerts_user_priority_unack ON public.alerts(user_id, priority, created_at DESC)
  WHERE acknowledged_at IS NULL;
CREATE INDEX idx_alerts_route_priority ON public.alerts(route, priority, created_at DESC);
CREATE INDEX idx_metrics_date_route ON public.performance_metrics(metric_date DESC, route);
CREATE INDEX idx_competitive_competitor_route_date ON public.competitive_events(competitor, route, detecte
CREATE INDEX idx_llm_usage_user_date ON public.llm_usage(user_id, created_at DESC);

-- Real-time subscriptions setup
ALTER PUBLICATION supabase_realtime ADD TABLE public.alerts;
ALTER PUBLICATION supabase_realtime ADD TABLE public.competitive_events;
ALTER PUBLICATION supabase_realtime ADD TABLE public.conversations;

-- Functions for updated_at timestamps
CREATE OR REPLACE FUNCTION public.handle_updated_at()
RETURNS TRIGGER AS $
BEGIN
  NEW.updated_at = NOW();
  RETURN NEW;
END;
$ LANGUAGE plpgsql;

-- Apply updated_at trigger to relevant tables
CREATE TRIGGER users_updated_at
  BEFORE UPDATE ON public.users
  FOR EACH ROW EXECUTE FUNCTION public.handle_updated_at();
```

Memory and Context Storage

```
javascript
```

```
// Context Management System
class ContextManager {
  private redis: Redis;
  private vectorDb: PineconeClient;

  async storeContext(sessionId: string, context: SessionContext) {
    // Store structured context in Redis
    await this.redis.setex(
      `context:${sessionId}`,
      3600, // 1 hour TTL
      JSON.stringify(context)
    );

    // Store semantic context in vector database
    const embedding = await this.generateEmbedding(context.summary);
    await this.vectorDb.upsert({
      vectors: [{
        id: sessionId,
        values: embedding,
        metadata: {
          userId: context.userId,
          timestamp: context.timestamp,
          type: context.type
        }
      }]
    });
  }

  async getRelevantContext(
    query: string,
    userId: string,
    limit: number = 5
  ): Promise<SessionContext[]> {

    const queryEmbedding = await this.generateEmbedding(query);

    const searchResults = await this.vectorDb.query({
      vector: queryEmbedding,
      filter: { userId: { $eq: userId } },
      topK: limit,
      includeMetadata: true
    });
    // Retrieve full context from Redis
    const contexts = await Promise.all(
      searchResults.matches.map(async (match) => {
        const contextData = await this.redis.get(`context:${match.id}`);
        return contextData ? JSON.parse(contextData) : null;
      })
    );

    return contexts.filter(Boolean);
  }
}
```

Development Setup

Replit Project Structure

```
velociti-platform/
├── .replit                 # Replit configuration
├── replit.nix              # Nix dependencies
├── package.json            # Node.js dependencies
├── tsconfig.json           # TypeScript configuration
├── tailwind.config.js      # Tailwind CSS setup
├── vite.config.ts          # Vite build configuration
├── prisma/
│   ├── schema.prisma       # Database schema
│   └── seed.ts             # Sample data seeding
├── src/
│   ├── client/             # React frontend
│   │   ├── components/     # React components
│   │   ├── pages/          # Page components
│   │   ├── hooks/          # Custom React hooks
│   │   ├── stores/         # Zustand stores
│   │   ├── types/          # TypeScript types
│   │   └── utils/          # Utility functions
│   ├── server/             # Express backend
│   │   ├── routes/         # API routes
│   │   ├── agents/         # AI agent implementations
│   │   ├── connectors/     # External API connectors
│   │   ├── jobs/           # Background job processors
│   │   ├── middleware/     # Express middleware
│   │   └── utils/          # Server utilities
│   └── shared/             # Shared types and utilities
├── data/                   # Sample EasyJet data
│   ├── competitive/        # Competitive intelligence data
│   ├── performance/        # Network performance data
│   └── routes.json         # EasyJet route configuration
└── docs/                   # Technical documentation
```

Environment Configuration

```
bash
```

```
# .env file for Replit Secrets
NODE_ENV=development
PORT=3000

# Database
DATABASE_URL="${REPLIT_DB_URL}"
REDIS_URL="${REPLIT_REDIS_URL}"

# AI Services
OPENAI_API_KEY="${OPENAI_API_KEY}"
WRITER_API_KEY="${WRITER_API_KEY}"
PINECONE_API_KEY="${PINECONE_API_KEY}"

# External APIs
DATABRICKS_HOST="${DATABRICKS_HOST}"
DATABRICKS_TOKEN="${DATABRICKS_TOKEN}"
DATABRICKS_WAREHOUSE_ID="${DATABRICKS_WAREHOUSE_ID}"

# Security
JWT_SECRET="${JWT_SECRET}"
SESSION_SECRET="${SESSION_SECRET}"

# Features
ENABLE_REAL_TIME=true
ENABLE_BACKGROUND_JOBS=true
LOG_LEVEL=debug
```

Development Scripts

```
json
```

```
{
  "scripts": {
    "dev": "concurrently \"npm run server:dev\" \"npm run client:dev\"",
    "server:dev": "tsx watch src/server/index.ts",
    "client:dev": "vite",
    "build": "npm run build:client && npm run build:server",
    "build:client": "vite build",
    "build:server": "tsc --project tsconfig.server.json",
    "db:generate": "prisma generate",
    "db:push": "prisma db push",
    "db:seed": "tsx prisma/seed.ts",
    "test": "vitest",
    "test:agents": "vitest src/server/agents/",
    "lint": "eslint src/ --ext .ts,.tsx",
    "type-check": "tsc --noEmit"
  }
}
```

API Design

RESTful API Endpoints

typescript

```
// Core API Routes
interface VelocitiAPI {
  // Authentication
  'POST /api/auth/login': LoginRequest => AuthResponse;
  'POST /api/auth/logout': {} => { success: boolean };
  'GET /api/auth/me': {} => UserProfile;

  // Intelligence Operations
  'GET /api/intelligence/briefing': BriefingRequest => MorningBriefing;
  'POST /api/intelligence/query': QueryRequest => IntelligenceResponse;
  'GET /api/intelligence/history': HistoryRequest => QueryHistory[];

  // Real-Time Alerts
  'GET /api/alerts': AlertsRequest => Alert[];
  'POST /api/alerts/:id/acknowledge': {} => { success: boolean };
  'PATCH /api/alerts/preferences': AlertPreferences => UserPreferences;

  // Data Integration
  'POST /api/data/query': DataQueryRequest => DataQueryResponse;
  'GET /api/data/schema': {} => DatabaseSchema;
  'POST /api/data/export': ExportRequest => ExportResponse;

  // Agent Management
  'GET /api/agents/status': {} => AgentStatus[];
  'POST /api/agents/:agentId/trigger': TriggerRequest => AgentResult;

  // Performance Metrics
  'GET /api/metrics/network': MetricsRequest => NetworkMetrics;
  'GET /api/metrics/competitive': CompetitiveRequest => CompetitiveMetrics;
}

// WebSocket Events
interface WebSocketEvents {
  // Client to Server
  'join-room': { userId: string };
  'query': QueryRequest;
  'acknowledge-alert': { alertId: string };

  // Server to Client
  'intelligence-update': VelocitiEvent;
  'alert': AlertEvent;
  'agent-result': AgentResult;
  'connection-status': { connected: boolean; agentsActive: number };
}
```

Agent API Implementation

typescript

```typescript
// Agent Base Class
abstract class BaseAgent implements IntelligenceAgent {
  protected name: string;
  protected specialization: string;
  protected eventBus: EventEmitter;

  constructor(name: string, specialization: string) {
    this.name = name;
    this.specialization = specialization;
    this.eventBus = new EventEmitter();
  }

  abstract async process(context: AgentContext): Promise<AgentResult>;

  protected async publishEvent(event: AgentEvent): Promise<void> {
    this.eventBus.emit('agent-event', event);
  }

  protected async getHistoricalContext(
    userId: string,
    lookback: number = 30
  ): Promise<HistoricalContext> {
    // Retrieve relevant historical data for context
    const metrics = await db.performanceMetrics.findMany({
      where: {
        userId,
        createdAt: {
          gte: new Date(Date.now() - lookback * 24 * 60 * 60 * 1000)
        }
      }
    });

    return { metrics, trends: this.calculateTrends(metrics) };
  }
}

// Competitive Intelligence Agent Implementation
class CompetitiveIntelligenceAgent extends BaseAgent {
  constructor() {
    super('competitive-intelligence', 'Competitive Analysis and Market Intelligence');
  }

  async process(context: AgentContext): Promise<AgentResult> {
```

```
  const startTime = Date.now();

  try {
   // Get competitive data
   const competitiveEvents = await this.getRecentCompetitiveEvents(context);

   // Analyze patterns
   const patterns = await this.analyzePatterns(competitiveEvents);

   // Generate insights
   const insights = await this.generateInsights(patterns, context);

   // Create recommendations
   const recommendations = await this.generateRecommendations(insights, context);

   return {
    agentId: this.name,
    confidence: this.calculateConfidence(insights),
    insights,
    recommendations,
    data: competitiveEvents,
    metadata: {
     processingTime: Date.now() - startTime,
     dataSourcesUsed: ['infare', 'oag', 'internal'],
     nextActions: this.suggestNextActions(recommendations)
    }
   };
  } catch (error) {
   throw new AgentError(`Competitive analysis failed: ${error.message}`);
  }
 }

 private async getRecentCompetitiveEvents(
  context: AgentContext
 ): Promise<CompetitiveEvent[]> {
  // Query competitive events from last 24 hours
  return await db.competitiveEvents.findMany({
   where: {
    detectedAt: {
     gte: new Date(Date.now() - 24 * 60 * 60 * 1000)
    }
   },
   orderBy: { detectedAt: 'desc' }
  });
  }

  private async analyzePatterns(events: CompetitiveEvent[]): Promise<Pattern[]> {
    // Pattern recognition logic
    const patterns: Pattern[] = [];

    // Group by competitor and route
    const grouped = groupBy(events, e => `${e.competitor}-${e.route}`);

    for (const [key, eventGroup] of Object.entries(grouped)) {
      if (eventGroup.length > 1) {
        patterns.push({
          type: 'repeated-action',
          competitor: eventGroup[0].competitor,
          route: eventGroup[0].route,
          frequency: eventGroup.length,
          confidence: this.calculatePatternConfidence(eventGroup)
        });
      }
    }

    return patterns;
  }
}
```

Testing Strategy

Testing Architecture

```typescript
```

```
// Test Configuration
import { describe, it, expect, beforeEach, afterEach } from 'vitest';
import { createTestClient } from './test-utils';

describe('Intelligence Agents', () => {
  let testClient: TestClient;
  let mockData: MockDataSet;

  beforeEach(async () => {
    testClient = await createTestClient();
    mockData = await loadMockData();
  });

  describe('CompetitiveIntelligenceAgent', () => {
    it('should detect Ryanair price changes correctly', async () => {
      const agent = new CompetitiveIntelligenceAgent();
      const context = {
        userId: 'test-user',
        sessionId: 'test-session',
        data: mockData.ryanairPriceChange,
        timestamp: new Date(),
        priority: 'high' as const
      };

      const result = await agent.process(context);

      expect(result.confidence).toBeGreaterThan(0.8);
      expect(result.insights).toHaveLength(1);
      expect(result.insights[0].type).toBe('competitive-threat');
      expect(result.recommendations).not.toBeEmpty();
    });

    it('should classify alert priority correctly', async () => {
      const highImpactChange = {
        competitor: 'Ryanair',
        route: 'LGW-BCN',
        oldPrice: 89.99,
        newPrice: 69.99,
        estimatedImpact: 45000
      };

      const classification = await classifyAlert(highImpactChange);
      expect(classification.priority).toBe('critical');
      expect(classification.slaMinutes).toBe(15);
    });
  });

  describe('Data Integration', () => {
    it('should handle Databricks API failures gracefully', async () => {
      const connector = new DatabricksConnector();

      // Mock API failure
      jest.spyOn(global, 'fetch').mockRejectedValue(new Error('Network error'));

      await expect(
        connector.executeQuery('SELECT * FROM routes', {})
      ).rejects.toThrow('Query failed: Network error');
    });

    it('should cache frequent queries', async () => {
      const connector = new DatabricksConnector();
      const query = 'SELECT route, avg_yield FROM performance WHERE date >= CURRENT_DATE - 7';

      // First call
      const result1 = await connector.executeQuery(query, {});

      // Second call should use cache
      const result2 = await connector.executeQuery(query, {});

      expect(result1).toEqual(result2);
      expect(connector.getCacheHitRate()).toBeGreaterThan(0);
    });
  });
});

// Integration Tests
describe('End-to-End Workflows', () => {
  describe('Morning Briefing Generation', () => {
    it('should generate complete briefing within SLA', async () => {
      const startTime = Date.now();

      const briefing = await generateMorningBriefing('test-user');

      const executionTime = Date.now() - startTime;
      expect(executionTime).toBeLessThan(30000); // 30 seconds
      expect(briefing.sections).toHaveLength(4);
      expect(briefing.criticalAlerts).toBeDefined();
    });
  });

  describe('Real-Time Alert Pipeline', () => {
    it('should process critical alerts within 15 minutes', async () => {
      const mockAlert = {
        type: 'competitive-price-change',
        competitor: 'Ryanair',
        route: 'LGW-BCN',
        impact: 55000,
        data: mockData.criticalPriceChange
      };

      const processingTime = await measureAlertProcessingTime(mockAlert);
      expect(processingTime).toBeLessThan(15 * 60 * 1000); // 15 minutes
    });
  });
});
```

Performance Optimization

Caching Strategy

```typescript
```

```
// Multi-Layer Caching Architecture
class CacheManager {
  private redisClient: Redis;
  private memoryCache: Map<string, CacheEntry>;
  private maxMemoryEntries = 1000;

  constructor() {
    this.redisClient = new Redis(process.env.REDIS_URL);
    this.memoryCache = new Map();
  }

  async get<T>(key: string): Promise<T | null> {
    // L1: Memory cache (fastest)
    const memoryEntry = this.memoryCache.get(key);
    if (memoryEntry && !this.isExpired(memoryEntry)) {
      return memoryEntry.data as T;
    }

    // L2: Redis cache (fast)
    const redisData = await this.redisClient.get(key);
    if (redisData) {
      const parsed = JSON.parse(redisData);
      this.setMemoryCache(key, parsed, 300); // 5 minutes
      return parsed as T;
    }

    return null;
  }

  async set<T>(key: string, data: T, ttlSeconds: number = 3600): Promise<void> {
    // Store in both layers
    this.setMemoryCache(key, data, Math.min(ttlSeconds, 300));
    await this.redisClient.setex(key, ttlSeconds, JSON.stringify(data));
  }

  private setMemoryCache<T>(key: string, data: T, ttlSeconds: number): void {
    // Implement LRU eviction
    if (this.memoryCache.size >= this.maxMemoryEntries) {
      const firstKey = this.memoryCache.keys().next().value;
      this.memoryCache.delete(firstKey);
    }

    this.memoryCache.set(key, {
      data,
      expiresAt: Date.now() + (ttlSeconds * 1000)
    });
  }
}

// Smart Query Caching
class QueryOptimizer {
  private cacheManager: CacheManager;
  private queryPatterns: Map<string, QueryPattern>;

  async executeOptimizedQuery(
    sql: string,
    context: QueryContext
  ): Promise<QueryResult> {

    // Generate cache key based on query and context
    const cacheKey = this.generateCacheKey(sql, context);

    // Check cache first
    const cached = await this.cacheManager.get<QueryResult>(cacheKey);
    if (cached && this.isCacheValid(cached, context)) {
      return { ...cached, fromCache: true };
    }

    // Execute query
    const result = await this.executeQuery(sql, context);

    // Cache based on query pattern
    const pattern = this.identifyQueryPattern(sql);
    const ttl = this.getCacheTTL(pattern);

    await this.cacheManager.set(cacheKey, result, ttl);

    return result;
  }

  private getCacheTTL(pattern: QueryPattern): number {
    switch (pattern.type) {
      case 'reference-data': return 24 * 3600; // 24 hours
      case 'performance-metrics': return 15 * 60; // 15 minutes
      case 'competitive-data': return 5 * 60; // 5 minutes
      case 'real-time-data': return 60; // 1 minute
      default: return 300; // 5 minutes
    }
  }
}
```

Database Optimization

```
sql
```

```sql
-- Performance Indexes
CREATE INDEX CONCURRENTLY idx_alerts_user_priority_date
ON alerts(user_id, priority, created_at DESC)
WHERE acknowledged_at IS NULL;

CREATE INDEX CONCURRENTLY idx_metrics_route_date_type
ON performance_metrics(route, metric_date DESC, metric_type)
INCLUDE (value, variance_pct);

CREATE INDEX CONCURRENTLY idx_competitive_events_recent
ON competitive_events(competitor, route, detected_at DESC)
WHERE detected_at >= CURRENT_DATE - INTERVAL '7 days';

-- Partitioning for Large Tables
CREATE TABLE performance_metrics_y2024 PARTITION OF performance_metrics
FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

CREATE TABLE performance_metrics_y2025 PARTITION OF performance_metrics
FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');

-- Materialized Views for Common Queries
CREATE MATERIALIZED VIEW route_performance_summary AS
SELECT
  route,
  DATE_TRUNC('day', metric_date) as day,
  AVG(CASE WHEN metric_type = 'yield' THEN value END) as avg_yield,
  AVG(CASE WHEN metric_type = 'load_factor' THEN value END) as avg_load_factor,
  COUNT(*) as data_points
FROM performance_metrics
WHERE metric_date >= CURRENT_DATE - INTERVAL '30 days'
GROUP BY route, DATE_TRUNC('day', metric_date);

CREATE UNIQUE INDEX ON route_performance_summary (route, day);

-- Refresh materialized views hourly
SELECT cron.schedule('refresh-route-summary', '0 * * * *',
  'REFRESH MATERIALIZED VIEW CONCURRENTLY route_performance_summary;');
```

Security Implementation

Authentication & Authorization

typescript

```
// JWT-based Authentication
class AuthService {
  private jwtSecret: string;
  private tokenExpiry = '24h';

  constructor() {
    this.jwtSecret = process.env.JWT_SECRET!;
  }

  async login(email: string, password: string): Promise<AuthResponse> {
    // Verify credentials (implement your auth logic)
    const user = await this.verifyCredentials(email, password);
    if (!user) {
      throw new UnauthorizedError('Invalid credentials');
    }

    // Generate JWT token
    const token = jwt.sign(
      {
        userId: user.id,
        email: user.email,
        role: user.role
      },
      this.jwtSecret,
      { expiresIn: this.tokenExpiry }
    );

    // Create session
    await this.createSession(user.id, token);

    return {
      token,
      user: {
        id: user.id,
        email: user.email,
        name: user.name,
        role: user.role
      },
      expiresAt: new Date(Date.now() + 24 * 60 * 60 * 1000)
    };
  }

  async verifyToken(token: string): Promise<TokenPayload> {
    try {
      const decoded = jwt.verify(token, this.jwtSecret) as TokenPayload;

      // Check if session is still valid
      const session = await this.getSession(decoded.userId, token);
      if (!session) {
        throw new UnauthorizedError('Session expired');
      }

      return decoded;
    } catch (error) {
      throw new UnauthorizedError('Invalid token');
    }
  }
}

// Role-Based Access Control
class RBACMiddleware {
  static requireRole(allowedRoles: UserRole[]) {
    return async (req: AuthenticatedRequest, res: Response, next: NextFunction) => {
      try {
        const token = req.headers.authorization?.replace('Bearer ', '');
        if (!token) {
          return res.status(401).json({ error: 'No token provided' });
        }

        const payload = await authService.verifyToken(token);

        if (!allowedRoles.includes(payload.role)) {
          return res.status(403).json({ error: 'Insufficient permissions' });
        }

        req.user = payload;
        next();
      } catch (error) {
        return res.status(401).json({ error: error.message });
      }
    };
  }

  static requirePermission(permission: string) {
    return async (req: AuthenticatedRequest, res: Response, next: NextFunction) => {
      const userPermissions = await this.getUserPermissions(req.user.userId);
      if (!userPermissions.includes(permission)) {
        return res.status(403).json({ error: 'Permission denied' });
      }

      next();
    };
  }
}

// Usage in routes
app.get('/api/intelligence/briefing',
  RBACMiddleware.requireRole(['analyst', 'manager']),
  async (req, res) => {
    // Generate briefing logic
  }
);

app.post('/api/agents/:agentId/trigger',
  RBACMiddleware.requireRole(['manager', 'executive']),
  RBACMiddleware.requirePermission('trigger-agents'),
  async (req, res) => {
    // Trigger agent logic
  }
);
```

Data Protection & Privacy

typescript

```
// Data Encryption Service
class EncryptionService {
  private algorithm = 'aes-256-gcm';
  private key: Buffer;

  constructor() {
    this.key = crypto.scryptSync(process.env.ENCRYPTION_KEY!, 'salt', 32);
  }

  encrypt(text: string): EncryptedData {
    const iv = crypto.randomBytes(16);
    const cipher = crypto.createCipher(this.algorithm, this.key);
    cipher.setAAD(Buffer.from('velociti-data'));

    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');

    const authTag = cipher.getAuthTag();

    return {
      encrypted,
      iv: iv.toString('hex'),
      authTag: authTag.toString('hex')
    };
  }

  decrypt(encryptedData: EncryptedData): string {
    const decipher = crypto.createDecipher(this.algorithm, this.key);
    decipher.setAAD(Buffer.from('velociti-data'));
    decipher.setAuthTag(Buffer.from(encryptedData.authTag, 'hex'));

    let decrypted = decipher.update(encryptedData.encrypted, 'hex', 'utf8');
    decrypted += decipher.final('utf8');

    return decrypted;
  }
}

// GDPR Compliance Service
class GDPRService {
  async handleDataExportRequest(userId: string): Promise<DataExport> {
    // Collect all user data across tables
    const userData = await this.collectUserData(userId);
  // Anonymize sensitive data
  const anonymized = this.anonymizeData(userData);

  // Generate export file
  const exportFile = await this.generateExportFile(anonymized);

  // Log the request
  await this.logDataExport(userId, exportFile.id);

  return exportFile;
}

async handleDataDeletionRequest(userId: string): Promise<DeletionResult> {
  const startTime = Date.now();

  try {
    // Soft delete user data (maintain referential integrity)
    await db.transaction(async (tx) => {
      await tx.users.update({
        where: { id: userId },
        data: {
          email: `deleted-${userId}@anonymized.local`,
          name: 'Deleted User',
          deletedAt: new Date()
        }
      });

      // Anonymize associated data
      await tx.conversations.updateMany({
        where: { session: { userId } },
        data: { content: '[REDACTED]' }
      });

      await tx.alerts.updateMany({
        where: { userId },
        data: { message: '[REDACTED]' }
      });
    });

    // Schedule hard deletion after retention period
    await this.scheduleHardDeletion(userId, 30); // 30 days

    return {
      success: true,
      deletedAt: new Date(),
      processingTime: Date.now() - startTime
    };
  } catch (error) {
    throw new GDPRError(`Data deletion failed: ${error.message}`);
  }
 }
}
```

Monitoring & Observability

Application Monitoring

```
typescript
```

```
// Comprehensive Logging System
class Logger {
  private winston: winston.Logger;
  private metrics: PrometheusRegistry;

  constructor() {
    this.winston = winston.createLogger({
      level: process.env.LOG_LEVEL || 'info',
      format: winston.format.combine(
        winston.format.timestamp(),
        winston.format.errors({ stack: true }),
        winston.format.json()
      ),
      transports: [
        new winston.transports.Console(),
        new winston.transports.File({ filename: 'velociti.log' })
      ]
    });

    this.setupMetrics();
  }

  private setupMetrics() {
    // Custom metrics for Velociti
    this.metrics = {
      agentExecutionTime: new prometheus.Histogram({
        name: 'velociti_agent_execution_seconds',
        help: 'Time spent executing intelligence agents',
        labelNames: ['agent_type', 'user_id', 'success']
      }),

      alertsGenerated: new prometheus.Counter({
        name: 'velociti_alerts_total',
        help: 'Total number of alerts generated',
        labelNames: ['priority', 'type', 'acknowledged']
      }),

      queryExecutionTime: new prometheus.Histogram({
        name: 'velociti_query_execution_seconds',
        help: 'Time spent executing data queries',
        labelNames: ['query_type', 'cache_hit']
      }),
    userSessions: new prometheus.Gauge({
      name: 'velociti_active_sessions',
      help: 'Number of active user sessions'
    })
  };
}

logAgentExecution(
  agentType: string,
  userId: string,
  executionTime: number,
  success: boolean
) {
  this.winston.info('Agent execution completed', {
    agentType,
    userId,
    executionTime,
    success,
    timestamp: new Date().toISOString()
  });

  this.metrics.agentExecutionTime
    .labels(agentType, userId, success.toString())
    .observe(executionTime / 1000);
}

logAlert(alert: Alert, acknowledged: boolean = false) {
  this.winston.info('Alert generated', {
    alertId: alert.id,
    priority: alert.priority,
    type: alert.alertType,
    userId: alert.userId,
    acknowledged
  });

  this.metrics.alertsGenerated
    .labels(alert.priority, alert.alertType, acknowledged.toString())
    .inc();
}
}

// Health Check System
class HealthChecker {
  private checks: Map<string, HealthCheck> = new Map();
```

```
constructor() {
  this.registerDefaultChecks();
}

private registerDefaultChecks() {
  // Database connectivity
  this.checks.set('database', {
    name: 'PostgreSQL Database',
    check: async () => {
      try {
        await db.$queryRaw`SELECT 1`;
        return { status: 'healthy', latency: 0 };
      } catch (error) {
        return { status: 'unhealthy', error: error.message };
      }
    }
  });

  // Redis connectivity
  this.checks.set('redis', {
    name: 'Redis Cache',
    check: async () => {
      try {
        const start = Date.now();
        await redis.ping();
        return { status: 'healthy', latency: Date.now() - start };
      } catch (error) {
        return { status: 'unhealthy', error: error.message };
      }
    }
  });

  // External API connectivity
  this.checks.set('writer-api', {
    name: 'Writer API',
    check: async () => {
      try {
        const response = await fetch('https://api.writer.com/v1/health');
        return {
          status: response.ok ? 'healthy' : 'degraded',
          latency: 0
        };
      } catch (error) {
        return { status: 'unhealthy', error: error.message };
      }
    }
  });

  // Agent system health
  this.checks.set('agents', {
    name: 'Intelligence Agents',
    check: async () => {
      const agentCoordinator = AgentCoordinator.getInstance();
      const activeAgents = await agentCoordinator.getActiveAgents();

      return {
        status: activeAgents.length > 0 ? 'healthy' : 'degraded',
        metadata: { activeAgents: activeAgents.length }
      };
    }
  });
}

async runHealthChecks(): Promise<HealthStatus> {
  const results = new Map<string, HealthCheckResult>();

  for (const [name, check] of this.checks) {
    try {
      const result = await Promise.race([
        check.check(),
        new Promise<HealthCheckResult>((_, reject) =>
          setTimeout(() => reject(new Error('Timeout')), 5000)
        )
      ]);
      results.set(name, result);
    } catch (error) {
      results.set(name, {
        status: 'unhealthy',
        error: error.message
      });
    }
  }

  // Determine overall status
  const statuses = Array.from(results.values()).map(r => r.status);
  const overallStatus = statuses.every(s => s === 'healthy') ? 'healthy' :
                    statuses.some(s => s === 'healthy') ? 'degraded' : 'unhealthy';
    return {
      status: overallStatus,
      timestamp: new Date().toISOString(),
      checks: Object.fromEntries(results),
      uptime: process.uptime()
    };
  }
}

// Performance Monitoring
class PerformanceMonitor {
  private performanceObserver: PerformanceObserver;

  constructor() {
    this.setupPerformanceTracking();
  }

  private setupPerformanceTracking() {
    this.performanceObserver = new PerformanceObserver((list) => {
      for (const entry of list.getEntries()) {
        if (entry.entryType === 'measure') {
          Logger.getInstance().winston.debug('Performance measurement', {
            name: entry.name,
            duration: entry.duration,
            startTime: entry.startTime
          });
        }
      }
    });

    this.performanceObserver.observe({ entryTypes: ['measure'] });
  }

  measureOperation<T>(operationName: string, operation: () => Promise<T>): Promise<T> {
    const startMark = `${operationName}-start`;
    const endMark = `${operationName}-end`;

    return new Promise(async (resolve, reject) => {
      try {
        performance.mark(startMark);
        const result = await operation();
        performance.mark(endMark);
        performance.measure(operationName, startMark, endMark);
        resolve(result);
      } catch (error) {
        performance.mark(endMark);
        performance.measure(operationName, startMark, endMark);
        reject(error);
      }
    });
  }
}
```

Deployment Configuration

Replit Production Deployment

```
yaml
```

```
# .github/workflows/deploy.yml
name: Deploy to Replit
on:
  push:
    branches: [main]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test

      - name: Build application
        run: npm run build

      - name: Deploy to Replit
        env:
          REPLIT_TOKEN: ${{ secrets.REPLIT_TOKEN }}
        run: |
          curl -X POST \
            -H "Authorization: Bearer $REPLIT_TOKEN" \
            -H "Content-Type: application/json" \
            -d '{"replId": "${{ secrets.REPL_ID }}", "action": "deploy"}' \
            https://replit.com/api/repls/deploy
```

Environment-Specific Configuration

```
typescript
```

```
// config/environments.ts
interface EnvironmentConfig {
  database: DatabaseConfig;
  redis: RedisConfig;
  ai: AIConfig;
  features: FeatureFlags;
  monitoring: MonitoringConfig;
}

const environments: Record<string, EnvironmentConfig> = {
  development: {
    database: {
      url: process.env.REPLIT_DB_URL!,
      ssl: false,
      poolSize: 5
    },
    redis: {
      url: process.env.REPLIT_REDIS_URL!,
      maxRetries: 3
    },
    ai: {
      provider: 'openai', // Use OpenAI for development
      apiKey: process.env.OPENAI_API_KEY!,
      model: 'gpt-4-turbo-preview'
    },
    features: {
      realTimeUpdates: true,
      backgroundJobs: true,
      advancedAnalytics: false
    },
    monitoring: {
      logLevel: 'debug',
      enableMetrics: true,
      healthCheckInterval: 30000
    }
  },

  production: {
    database: {
      url: process.env.DATABASE_URL!,
      ssl: true,
      poolSize: 20
    },
    redis: {
      url: process.env.REDIS_URL!,
      maxRetries: 5
    },
    ai: {
      provider: 'writer', // Use Writer API for production
      apiKey: process.env.WRITER_API_KEY!,
      model: 'palmyra-x-5'
    },
    features: {
      realTimeUpdates: true,
      backgroundJobs: true,
      advancedAnalytics: true
    },
    monitoring: {
      logLevel: 'info',
      enableMetrics: true,
      healthCheckInterval: 60000
    }
  }
};

export const config = environments[process.env.NODE_ENV || 'development'];
```

Development Workflow

Getting Started

```
bash
```

```
# 1. Clone the repository in Replit
# 2. Install dependencies
npm install

# 3. Set up environment variables in Replit Secrets
# Required secrets:
# - DATABASE_URL
# - REDIS_URL
# - OPENAI_API_KEY (for development)
# - WRITER_API_KEY (for production)
# - JWT_SECRET

# 4. Initialize database
npm run db:push
npm run db:seed

# 5. Start development server
npm run dev
```

Development Commands

```json
{
  "scripts": {
    "dev": "concurrently \"npm run server:dev\" \"npm run client:dev\"",
    "server:dev": "tsx watch src/server/index.ts",
    "client:dev": "vite --port 3001",
    "build": "npm run build:client && npm run build:server",
    "test": "vitest",
    "test:watch": "vitest --watch",
    "test:agents": "vitest src/server/agents/ --reporter=verbose",
    "db:studio": "prisma studio",
    "db:migrate": "prisma migrate dev",
    "db:reset": "prisma migrate reset",
    "lint": "eslint . --ext .ts,.tsx --fix",
    "format": "prettier --write .",
    "type-check": "tsc --noEmit"
  }
}
```

Code Quality & Standards

typescript

```typescript
// .eslintrc.js
module.exports = {
  extends: [
    '@typescript-eslint/recommended',
    'plugin:react/recommended',
    'plugin:react-hooks/recommended'
  ],
  rules: {
    '@typescript-eslint/no-unused-vars': 'error',
    '@typescript-eslint/explicit-function-return-type': 'warn',
    'react/react-in-jsx-scope': 'off',
    'no-console': 'warn'
  }
};

// prettier.config.js
module.exports = {
  semi: true,
  trailingComma: 'es5',
  singleQuote: true,
  printWidth: 80,
  tabWidth: 2
};
```

This comprehensive Technical Requirements Document provides the complete blueprint for implementing Velociti on Replit. The architecture balances rapid development capabilities with production-ready patterns, ensuring the platform can scale from prototype to enterprise deployment while maintaining the flexibility needed for AI-driven intelligence operations.