Land Memory System - Detailed Implementation

1. Core Components Implementation

1.1 Type System (`types.ts`)

Enums

- Implement categorical data types using TypeScript enums
- Ensure exhaustive pattern matching
- Document valid value ranges

```typescript
export enum PlotSize {
    Nano = 'Nano',
    // ...
}
```

Interfaces

- Define strict type boundaries
- Use nested structures for complex data
- Include JSDoc documentation

```typescript
export interface LandPlotMetadata {
    rank: number;
    // ...
}
```

1.2 Database Adapter (`land_database_adapter.ts`)

Query Building

```typescript
// Base query structure
let sql = `SELECT * FROM memories WHERE type = $1`;
const values: any[] = [LAND_TABLE];

// Dynamic parameter addition
if (params.neighborhoods?.length) {
    sql += ` AND content->'metadata'->>'neighborhood' =
ANY($${++paramCount}::text[])`;
    values.push(params.neighborhoods);
}
```

Error Handling

```
try {
    const { rows } = await this.query(sql, values);
    return rows.map(row => ({
        ...row,
        content: typeof row.content === 'string' ? JSON.parse(row.content)
: row.content
    }));
} catch (error) {
    elizaLogger.error('Error in searchLandByMetadata:', {
        error: error instanceof Error ? error.message : String(error),
        params
    });
    throw error;
}
```

1.3 Memory System (`land_memory_system.ts`)

CSV Processing

```
private generatePlotDescription(plot: any): string {
    return `${plot.Name} is a ${plot['Plot Size']} ...`;
}
```

Search Implementation

```
async searchProperties(
    query: string,
    metadata: Partial<LandSearchParams> = {},
    limit: number = DEFAULT_MATCH_COUNT
): Promise<LandPlotMemory[]>
```

2. Database Schema

2.1 Memories Table

```
CREATE TABLE memories (
    id UUID PRIMARY KEY,
    type TEXT NOT NULL,
    room_id TEXT NOT NULL,
    agent_id TEXT NOT NULL,
    content JSONB,
```

```
    embedding vector(1536)
);
```

2.2 Indexes

```sql
CREATE INDEX idx_memories_type ON memories(type);
CREATE INDEX idx_memories_content ON memories USING gin(content);
CREATE INDEX idx_memories_embedding ON memories USING ivfflat (embedding
vector_cosine_ops);
```

3. Query Optimization

3.1 Metadata Queries

- Use JSON containment operators
- Leverage GIN indexes
- Implement parameter sanitization

3.2 Vector Search

- Use HNSW index for embeddings
- Implement similarity thresholds
- Optimize result count

4. Error Handling Strategy

4.1 Error Types

```
enum ErrorType {
    DATABASE_ERROR = 'DATABASE_ERROR',
    VALIDATION_ERROR = 'VALIDATION_ERROR',
    EMBEDDING_ERROR = 'EMBEDDING_ERROR'
}
```

4.2 Error Logging

```
elizaLogger.error('Error Type:', {
    type: ErrorType.DATABASE_ERROR,
    details: error.message,
    context: {
        query,
        params
    }
});
```

5. Performance Considerations

5.1 Query Optimization

- Use prepared statements
- Implement connection pooling
- Optimize JSON operations

5.2 Memory Management

- Implement result pagination
- Use streaming for large datasets
- Optimize embedding storage

6. Security Measures

6.1 Input Validation

- Sanitize SQL inputs
- Validate JSON structure
- Check parameter bounds

6.2 Access Control

- Implement role-based access
- Validate agent permissions
- Log access attempts

7. Monitoring and Logging

7.1 Metrics

- Query performance
- Memory usage
- Error rates

7.2 Logging

- Request/response cycles
- Error stacks
- Performance data

8. Configuration Management

8.1 Environment Variables

```
const config = {
    database: {
        host: process.env.DB_HOST,
        port: process.env.DB_PORT,
```

```
        name: process.env.DB_NAME
    },
    search: {
        defaultThreshold: process.env.SEARCH_THRESHOLD,
        maxResults: process.env.MAX_RESULTS
    }
};
```

8.2 Feature Flags

```
const features = {
    enableVectorSearch: true,
    useCache: process.env.NODE_ENV === 'production',
    debugMode: process.env.DEBUG === 'true'
};
```

9. Testing Strategy

9.1 Unit Tests

- Type validation
- Query building
- Error handling

9.2 Integration Tests

- Database operations
- Search functionality
- CSV processing

9.3 Performance Tests

- Query benchmarks
- Memory usage
- Concurrent operations