Neuro-CLI

(Orbit)

Complete Technical Documentation

AI-Powered Command-Line Interface & Web Platform

Technology Stack

Next.js 16 • React 19 • Express 5.1
PostgreSQL • Prisma ORM • Google AI SDK

Mausam Kar

Version 1.0.0 | 2026 Edition

Neuro-CLI (Orbit) - Complete Technical Documentation
Version 1.0.0 | January 2026

Technologies:

- Frontend: Next.js 16.0, React 19.2, TailwindCSS 4
- Backend: Express 5.1, Node.js (Latest)
- Database: PostgreSQL with Prisma ORM 6.18
- AI: Google AI SDK (@ai-sdk/google)
- CLI: Commander.js, Clack, Inquirer
- Auth: Better-Auth with Device Flow
- UI: Radix UI (50+ components)

Documentation Built With: LaTeX

For contributions, issues, or questions, please visit the project repository.

Dedicated to developers who believe in the power of AI-assisted development and seamless user experiences.

"The command line is where productivity meets power."

Contents

Preface

Neuro-CLI (Orbit) represents the convergence of traditional command-line power with modern AI capabilities. This documentation serves as your comprehensive guide to understanding, deploying, and extending this innovative platform.

What is Neuro-CLI?

Neuro-CLI is a full-stack AI platform that provides multiple interfaces for seamless AI interaction:

- **Orbit CLI** - A powerful terminal interface for developers who prefer command-line workflows
- **Web Application** - A beautiful Next.js-based platform for browser-based AI interactions
- **Unified Authentication** - Device-flow authentication ensuring security across all platforms
- **Persistent Conversations** - PostgreSQL-backed storage for conversation history and context

Why This Documentation?

This manual provides:

Documentation Objectives

- Complete architectural understanding of the platform
- Step-by-step setup and configuration guides
- Comprehensive API and CLI reference
- Best practices for development and deployment
- Troubleshooting guides for common issues

Who Should Use This?

This documentation is designed for:

- **Developers** implementing or extending the platform
- **DevOps Engineers** deploying and maintaining the system
- **System Administrators** managing user access and data
- **Technical Evaluators** assessing the platform's capabilities
- **Students** learning full-stack AI development

Document Structure

Chapter Overview

Chapter 1	System architecture and core concepts
Chapter 2	Installation and configuration
Chapter 3	Web application (Next.js client)
Chapter 4	API server (Express backend)
Chapter 5	Orbit CLI usage and development
Chapter 6	Database schema and management
Chapter 7	Authentication and security
Chapter 8	Deployment and production

Conventions Used

Throughout this documentation:

- **Bold text** indicates important concepts or UI elements
- `Monospace text` represents code, commands, or file paths
- `Inline code` shows specific values or short snippets

Mausam Kar
January 2026

Chapter 1

System Overview

Neuro-CLI (Orbit) is a modern, full-stack AI platform that seamlessly bridges command-line efficiency with web-based accessibility. This chapter provides a comprehensive overview of the system architecture, core features, and technology stack.

1.1 Platform Architecture

The platform follows a three-tier architecture with clear separation of concerns:

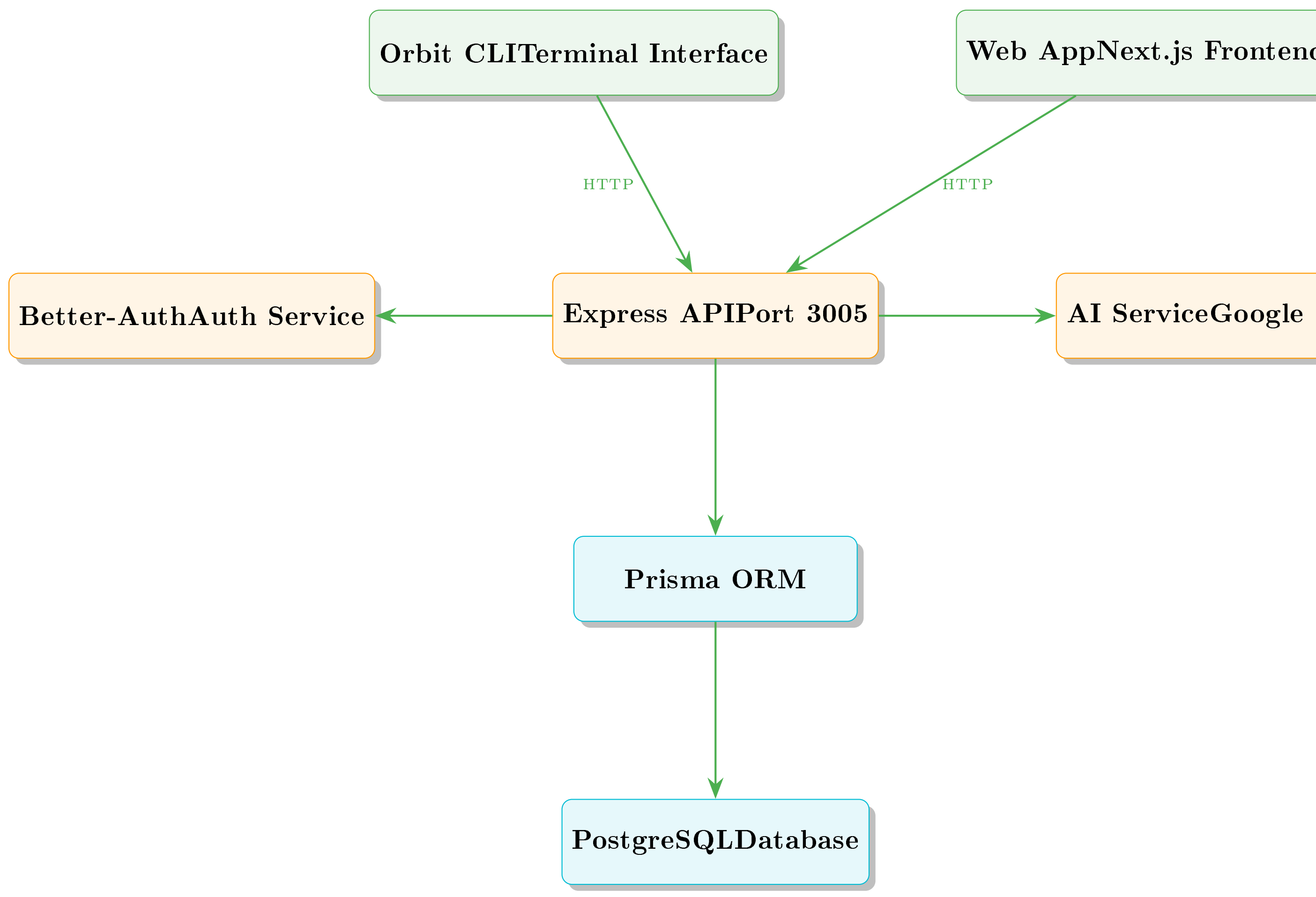

Figure 1.1: High-Level System Architecture

1.1.1 Layer Responsibilities

Client Layer

Purpose: User interaction and experience

- **Orbit CLI**: Command-line interface for terminal-based AI interactions
- **Web Application**: Browser-based platform with rich UI components
- **Shared Authentication**: Unified device-flow authentication across both clients

Server Layer

Purpose: Business logic and API endpoints

- **Express API**: RESTful endpoints for client communication
- **Better-Auth**: Secure device-flow authentication service
- **AI Service**: Google AI SDK integration for conversational AI
- **Session Management**: Token-based session handling

Data Layer

Purpose: Persistent data storage and management

- **PostgreSQL**: Relational database for structured data
- **Prisma ORM**: Type-safe database access and migrations
- **Schema Management**: Automated migrations and versioning

1.2 Core Features

1.2.1 Multi-Interface Access

Neuro-CLI provides flexibility through multiple access points:

Interface	Use Case	Key Features
Orbit CLI	Quick terminal access, automation, scripting	Markdown rendering, interactive prompts, persistent sessions
Web App	Rich UI interactions, visual feedback, collaboration	50+ UI components, dark/light themes, real-time updates
Both	Synchronized experience	Shared authentication, conversation sync, unified data

Table 1.1: Interface Comparison

1.2.2 Authentication System

Device Flow Authentication

The platform implements OAuth 2.0 Device Authorization Grant for secure, user-friendly authentication:

1. User initiates login from CLI or web
2. System generates unique device code and user code
3. User visits web interface with user code
4. User approves device on authenticated web session
5. CLI/client receives access token
6. Session established with expiration tracking

1.2.3 AI Capabilities

Mode	Type	Description
Chat	Conversational	Free-form conversation with AI assistant
Tool	Function-calling	AI can invoke defined tools and functions
Agent	Autonomous	AI acts as independent agent with goals

Table 1.2: AI Interaction Modes

1.3 Technology Stack

1.3.1 Frontend Technologies

Next.js Web Application

Framework	Next.js 16.0 with App Router
UI Library	React 19.2 with latest features
Styling	TailwindCSS 4 with custom design system
Components	Radix UI primitives (50+ components)
Forms	React Hook Form + Zod validation
Charts	Recharts for data visualization
Theme	Next-themes for dark/light mode
Auth Client	Better-Auth client integration

1.3.2 Backend Technologies

Express.js API Server

Runtime	Node.js (Latest LTS)
Framework	Express 5.1
Database	PostgreSQL (14.x or higher)
ORM	Prisma 6.18
Authentication	Better-Auth with device flow
AI SDK	Google AI SDK (@ai-sdk/google, ai)
Process Manager	PM2 (production)

1.3.3 CLI Technologies

Orbit CLI Tools

Framework	Commander.js for command structure
Prompts	Clack for beautiful interactive prompts
Input	Inquirer for complex user input
Styling	Chalk for colored terminal output
ASCII Art	Figlet for branded headers
Boxes	Boxen for framed messages
Spinners	Ora for loading animations
Markdown	Terminal markdown renderer

1.4 Data Flow

1.4.1 Conversation Lifecycle

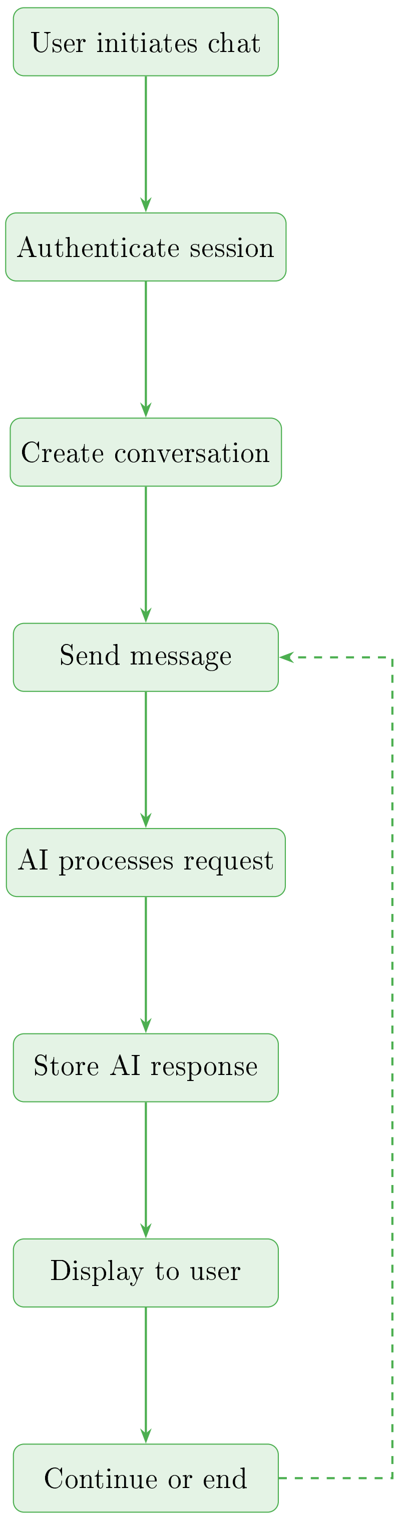

Figure 1.2: Conversation Flow Diagram

1.5 Project Structure

Directory Layout

```
Neuro-CLI/
 client/                    # Next.js Web Application
    app/                   # App Router pages
       (auth)/           # Auth routes
       approve/          # Device approval
       device/           # Code entry
       page.tsx          # Home page
    components/           # UI components
       ui/              # Radix primitives
    hooks/                # Custom React hooks
    lib/                  # Utils & helpers
    package.json

 server/                   # Express.js Backend
    src/
       cli/             # Orbit CLI
          main.js     # CLI entry
          commands/   # Commands
          chat/       # Chat modes
          ai/         # AI services
       config/         # Configuration
       lib/            # Shared libs
       services/       # Business logic
       index.js        # API server
    prisma/
       schema.prisma   # DB schema
    package.json

 README.md
```

1.6 Key Design Decisions

1.6.1 Why Device Flow Authentication?

Device Flow Benefits

Device flow authentication was chosen for several compelling reasons:

1. **CLI-Friendly**: No need to embed web browsers in CLI
2. **Secure**: No credential handling in terminal
3. **User Experience**: Simple code entry instead of complex flows
4. **Cross-Platform**: Works identically on all platforms
5. **Separation**: Auth happens in trusted browser environment

1.6.2 Why Next.js + Express?

Technology	Rationale
Next.js	Server-side rendering, excellent developer experience, built-in routing, React 19 support
Express	Mature ecosystem, simple API design, perfect for CLI back-end, widespread adoption
Separation	Allows independent scaling, specialized optimization, clear responsibilities

Table 1.3: Architecture Rationale

Chapter 2

Installation & Setup

This chapter provides comprehensive instructions for setting up the Neuro-CLI platform on your local development environment or production server.

2.1 Prerequisites

System Requirements

Ensure your system meets the following requirements before proceeding:

Tool	Version	Download
Node.js	18.x or higher	`https://nodejs.org/`
npm	9.x or higher	Included with Node.js
PostgreSQL	14.x or higher	`https://www.postgresql.org/`
Git	Latest	`https://git-scm.com/`

2.1.1 Verifying Prerequisites

Run these commands to verify installations:

Version Checks

```
# Check Node.js version
node --version
# Expected: v18.x.x or higher

# Check npm version
npm --version
# Expected: 9.x.x or higher

# Check PostgreSQL version
psql --version
# Expected: 14.x or higher

# Check Git version
git --version
```

2.2 Repository Setup

2.2.1 Clone the Repository

Git Clone

```
# Clone the repository
git clone <repository-url>
cd Neuro-CLI

# Verify structure
ls -la
# You should see: client/, server/, README.md
```

2.2.2 Install Dependencies

Server Dependencies

```
cd server
npm install

# This installs:
# - Express 5.1 and middleware
# - Prisma 6.18 and PostgreSQL driver
# - Better-Auth with device flow
# - Google AI SDK
# - Commander.js, Clack, Inquirer
# - Chalk, Figlet, Boxen, Ora
```

Client Dependencies

```
cd ../client
npm install

# This installs:
# - Next.js 16.0 and React 19.2
# - TailwindCSS 4
# - Radix UI components (50+)
# - React Hook Form + Zod
# - Better-Auth client
# - Recharts, Next-themes
```

2.3 Database Configuration

2.3.1 PostgreSQL Setup

Database Creation

Create a new PostgreSQL database for the project:

```
# Connect to PostgreSQL
psql -U postgres

# Create database
CREATE DATABASE neurocli;

# Create user (optional)
CREATE USER neurocli_user WITH PASSWORD 'your_password';

# Grant privileges
GRANT ALL PRIVILEGES ON DATABASE neurocli TO neurocli_user;

# Exit psql
\q
```

2.3.2 Prisma Configuration

Database Setup Commands

```
cd server

# Generate Prisma Client
npx prisma generate

# Push schema to database
npx prisma db push

# Open Prisma Studio (optional - database GUI)
npx prisma studio
```

2.4 Environment Variables

2.4.1 Server Environment

Create `server/.env` file:

server/.env Configuration

```
# Database Connection
DATABASE_URL="postgresql://user:password@localhost:5432/neurocli"

# Better-Auth Configuration
BETTER_AUTH_SECRET="your-32-character-secret-key-here"
BETTER_AUTH_URL="http://localhost:3005"

# Google AI SDK
GOOGLE_AI_API_KEY="your-google-ai-api-key"

# Server Configuration
PORT=3005
NODE_ENV=development

# CORS Origins (comma-separated)
CORS_ORIGINS="http://localhost:3000,http://localhost:3005"

# Session Configuration
SESSION_EXPIRES_IN=604800  # 7 days in seconds
DEVICE_CODE_EXPIRES_IN=600  # 10 minutes in seconds
```

Security Notice

Never commit .env files to version control!
The `BETTER_AUTH_SECRET` should be a cryptographically secure random string. Generate one using:

```
node -e "console.log(require('crypto').randomBytes(32).toString('hex'))"
```

2.4.2 Client Environment

Create `client/.env.local` file:

client/.env.local Configuration

```
# API Server URL
NEXT_PUBLIC_API_URL="http://localhost:3005"

# Better-Auth Client
NEXT_PUBLIC_BETTER_AUTH_URL="http://localhost:3005"

# Application Settings
NEXT_PUBLIC_APP_NAME="Neuro-CLI"
NEXT_PUBLIC_APP_VERSION="1.0.0"
```

2.5 Running the Application

2.5.1 Development Mode

You'll need three terminal windows:

Terminal 1: API Server

```
cd server
npm run dev

# Server starts on http://localhost:3005
# Watch for: "Server running on port 3005"
```

Terminal 2: Web Application

```
cd client
npm run dev

# Next.js starts on http://localhost:3000
# Open browser to http://localhost:3000
```

Terminal 3: CLI Usage

```
cd server
npm run cli -- --help

# Available commands:
# - login    : Authenticate via device flow
# - logout   : End current session
# - whoami   : Display user information
# - wakeUp   : Start AI chat session
```

2.5.2 Verifying Installation

Component	URL/Command	Expected Result
API Server	`http://localhost:3005`	JSON response
Web App	`http://localhost:3000`	Home page loads
CLI	`npm run cli - -help`	Commands listed
Database	`npx prisma studio`	GUI opens

Table 2.1: Verification Checklist

2.6 Initial Configuration

2.6.1 Create Admin User (Optional)

If you need to manually create an admin user:

Database Seed Script

```
# server/prisma/seed.js
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

async function main() {
  const admin = await prisma.user.create({
    data: {
      email: 'admin@neurocli.com',
      name: 'Admin User',
      emailVerified: true,
    },
  });
  console.log('Admin user created:', admin);
}

main().catch(console.error).finally(() => prisma.$disconnect());
```

Run with: `node prisma/seed.js`

2.7 Troubleshooting

2.7.1 Common Installation Issues

Database Connection Errors

Error: `Can't reach database server`
Solutions:

1. Verify PostgreSQL is running: `pg_ctl status`
2. Check DATABASE_URL format
3. Ensure database exists: `psql -l | grep neurocli`
4. Verify user permissions

Port Already in Use

Error: `Port 3005 already in use`
Solutions:

```
# Find process using port
lsof -i :3005

# Kill process (macOS/Linux)
kill -9 <PID>

# Windows
netstat -ano | findstr :3005
taskkill /PID <PID> /F
```

Module Not Found

Error: `Cannot find module '...'`
Solutions:

1. Delete `node_modules` and `package-lock.json`
2. Run `npm install` again
3. Clear npm cache: `npm cache clean -force`
4. Verify Node.js version compatibility

2.7.2 Prisma Issues

Prisma Client Not Generated

Error: `@prisma/client did not initialize yet`
Solution:

```
npx prisma generate
```

Migration Conflicts

Error: `Migration engine error`

Solution:

```
# Reset database (WARNING: deletes all data)
npx prisma migrate reset

# Or use db push for development
npx prisma db push --accept-data-loss
```

Chapter 3

Web Application

The Next.js web application provides a modern, accessible interface for interacting with Neuro-CLI through a browser. This chapter covers the client architecture, components, and development workflow.

3.1 Architecture Overview

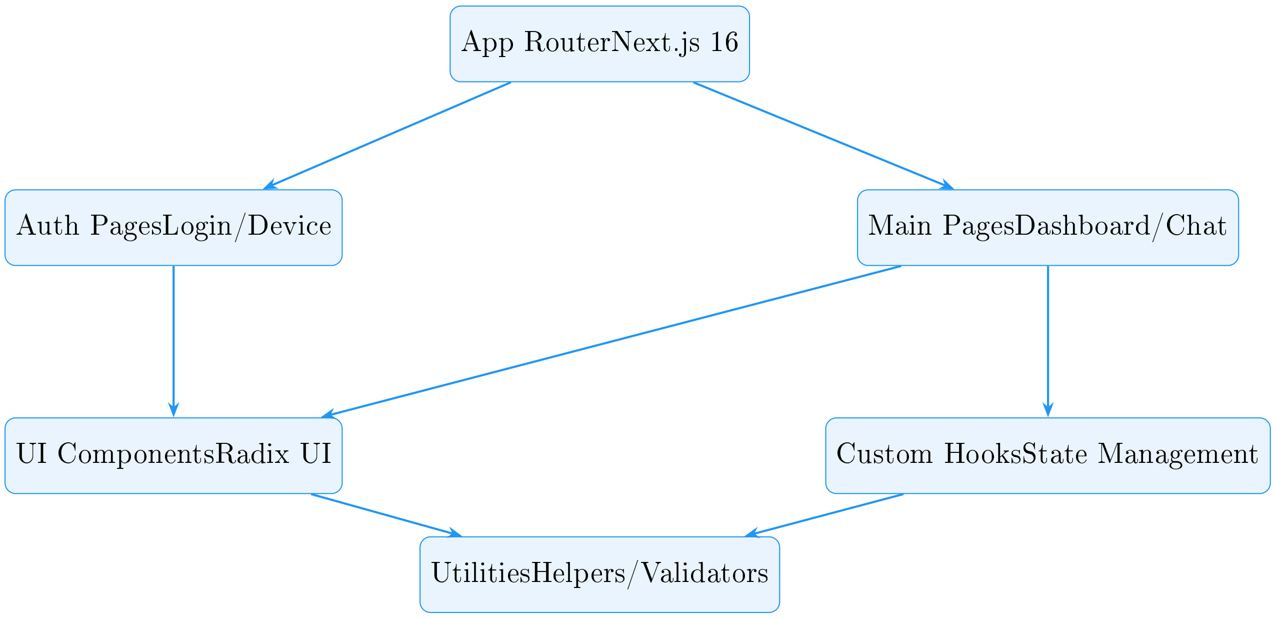

Figure 3.1: Client Application Structure

3.2 Key Features

3.2.1 App Router Pages

Route	File	Purpose
`/`	`app/page.tsx`	Home page/dashboard
`/device`	`app/device/page.tsx`	Device code entry
`/approve`	`app/approve/page.tsx`	Device approval flow
`/(auth)/login`	`app/(auth)/login/page.tsx`	User login
`/(auth)/signup`	`app/(auth)/signup/page.tsx`	User registration

Table 3.1: Application Routes

3.2.2 UI Component Library

50+ Radix UI Components

The application includes a comprehensive set of accessible UI primitives:

- **Layout**: Accordion, Tabs, Separator, Scroll Area
- **Overlays**: Dialog, Dropdown Menu, Popover, Tooltip
- **Forms**: Input, Select, Checkbox, Radio, Switch
- **Feedback**: Alert, Toast, Progress, Spinner
- **Navigation**: Navigation Menu, Breadcrumb
- **Data Display**: Table, Card, Badge, Avatar

3.3 Development Workflow

3.3.1 Running the Dev Server

Development Commands

```
cd client

# Start development server with hot reload
npm run dev

# Build for production
npm run build

# Start production server
npm start

# Run linting
npm run lint

# Format code
npm run format
```

3.3.2 Project Structure

Client Directory Layout

```
client/
 app/                         # Next.js App Router
    (auth)/                 # Auth route group
       login/
       signup/
    approve/                # Device approval
    device/                 # Device code entry
    layout.tsx              # Root layout
    page.tsx                # Home page
    globals.css             # Global styles

 components/                 # React components
    ui/                    # Radix UI primitives
       accordion.tsx
       button.tsx
       dialog.tsx
       ... (50+ components)
    auth/                  # Auth components
    chat/                  # Chat UI
    shared/                # Shared components

 hooks/                      # Custom React hooks
    use-auth.ts
    use-chat.ts
    use-toast.ts

 lib/                        # Utilities
    auth.ts               # Auth helpers
    api.ts                # API client
    utils.ts              # General utilities
    validators.ts         # Zod schemas

 public/                     # Static assets
 tailwind.config.ts          # Tailwind configuration
 next.config.js              # Next.js configuration
 package.json
```

3.4 Authentication Flow

3.4.1 Device Flow Implementation

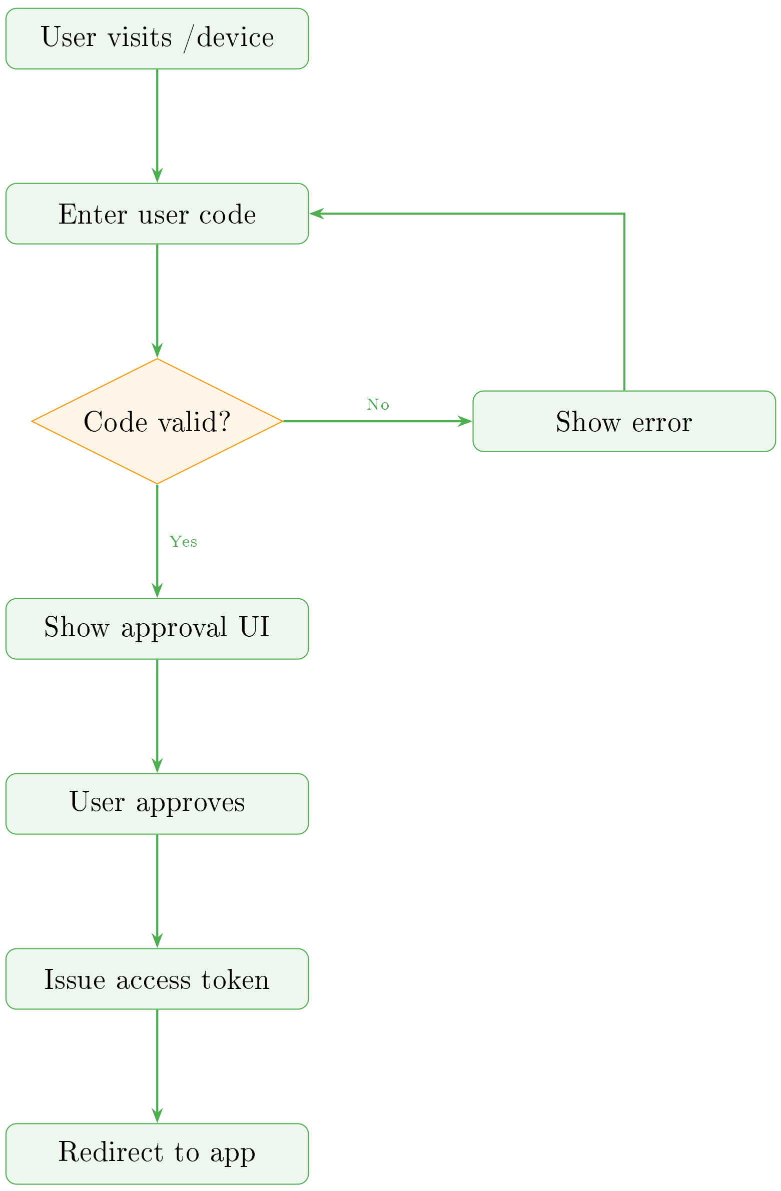

Figure 3.2: Device Approval Flow

3.4.2 Authentication Components

Key Auth Components

DeviceCodeEntry	Input form for 6-digit user code
ApprovalDialog	Confirmation dialog for device approval
SessionProvider	React context for auth state
ProtectedRoute	HOC for route protection
LoginButton	Trigger authentication flow

3.5 Styling System

3.5.1 TailwindCSS Configuration

Custom Theme Configuration

```
// tailwind.config.ts
export default {
  content: ['./app/**/*.{ts,tsx}', './components/**/*.{ts,tsx}'],
  theme: {
    extend: {
      colors: {
        primary: {
          DEFAULT: '#4CAF50',
          light: '#81C784',
          dark: '#388E3C',
        },
        secondary: {
          DEFAULT: '#2196F3',
          light: '#64B5F6',
          dark: '#1976D2',
        },
        accent: {
          DEFAULT: '#FF9800',
          light: '#FFB74D',
          dark: '#F57C00',
        },
      },
      fontFamily: {
        sans: ['Inter', 'system-ui', 'sans-serif'],
        mono: ['Fira Code', 'monospace'],
      },
    },
  },
  plugins: [
    require('@tailwindcss/forms'),
    require('@tailwindcss/typography'),
  ],
}
```

3.5.2 Dark Mode Support

Theme Switching

The application supports automatic theme switching based on system preferences or user selection:

```
// app/layout.tsx
import { ThemeProvider } from 'next-themes'

export default function RootLayout({ children }) {
  return (
    <html suppressHydrationWarning>
      <body>
        <ThemeProvider attribute="class" defaultTheme="system">
          {children}
        </ThemeProvider>
      </body>
    </html>
  )
}
```

3.6 API Integration

3.6.1 API Client Setup

API Client Configuration

```
// lib/api.ts
const API_BASE = process.env.NEXT_PUBLIC_API_URL

export const api = {
  // Authentication
  async login(credentials) {
    return fetch(‘${API_BASE}/api/auth/login‘, {
      method: ’POST’,
      headers: { ’Content-Type’: ’application/json’ },
      body: JSON.stringify(credentials),
    })
  },

  // Device flow
  async getDeviceCode() {
    return fetch(‘${API_BASE}/api/auth/device/code‘, {
      method: ’POST’,
    })
  },

  // Session management
  async getSession(token) {
    return fetch(‘${API_BASE}/api/me‘, {
      headers: { Authorization: ‘Bearer ${token}‘ },
    })
  },
}
```

3.7 Performance Optimization

Optimization Techniques

- **Code Splitting**: Automatic route-based splitting
- **Image Optimization**: Next.js Image component
- **Font Optimization**: next/font for Google Fonts
- **Static Generation**: Pre-render pages at build time
- **React Server Components**: Reduce client bundle size
- **Lazy Loading**: Dynamic imports for heavy components

Chapter 4

API Server

The Express.js server provides RESTful API endpoints, manages authentication, handles AI interactions, and coordinates database operations. This chapter details the backend architecture and API reference.

4.1 Server Architecture

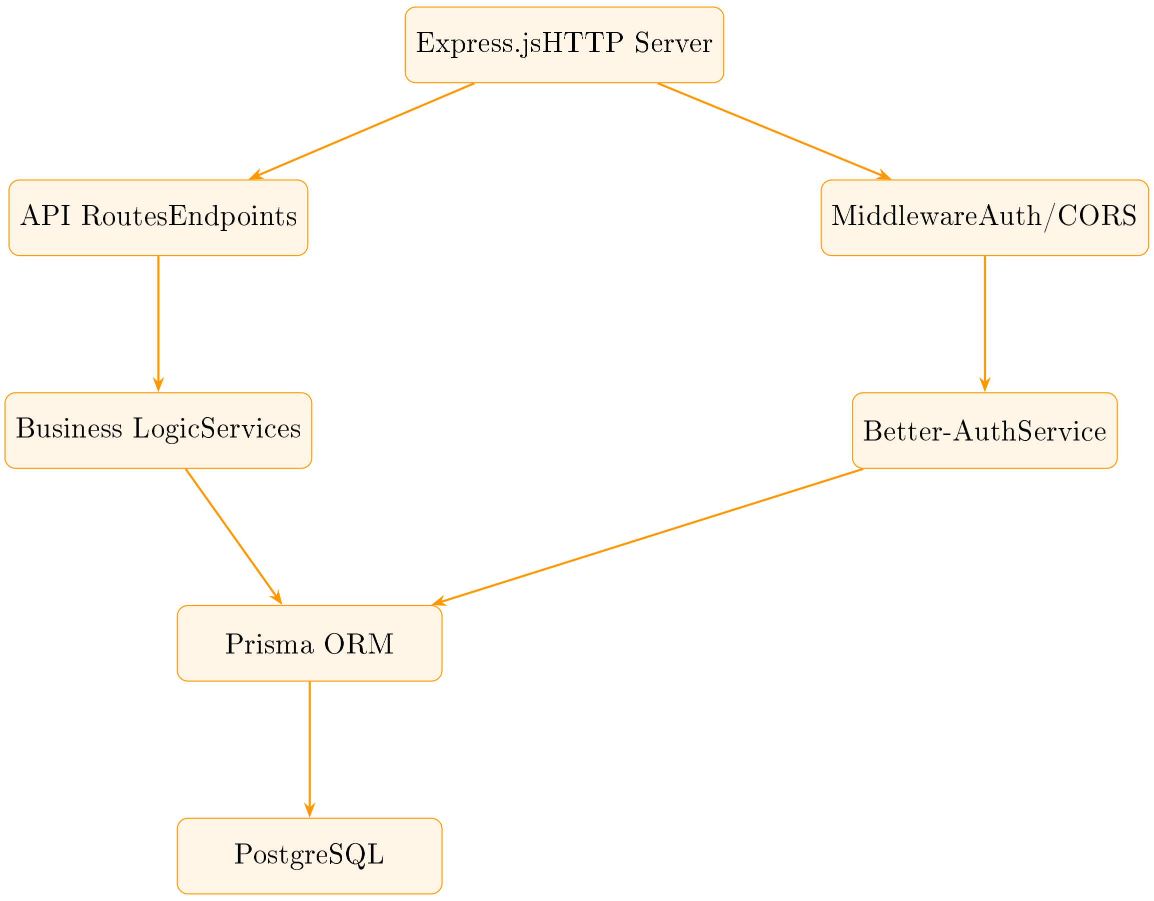

Figure 4.1: Server Architecture Layers

4.2 API Endpoints

4.2.1 Authentication Endpoints

Method	Endpoint	Description
POST	`/api/auth/login`	Standard email/password login
POST	`/api/auth/signup`	Create new user account
POST	`/api/auth/device/code`	Request device code
POST	`/api/auth/device/authorize`	Approve device
POST	`/api/auth/device/token`	Exchange code for token
POST	`/api/auth/logout`	End session

Table 4.1: Authentication API Endpoints

4.2.2 Session Management

Method	Endpoint	Description
GET	`/api/me`	Get current user (Bearer token)
GET	`/api/me/:token`	Get session by token (deprecated)
GET	`/api/sessions`	List user sessions
DELETE	`/api/sessions/:id`	Revoke session

Table 4.2: Session Management Endpoints

4.2.3 Conversation Endpoints

Method	Endpoint	Description
GET	`/api/conversations`	List user conversations
POST	`/api/conversations`	Create conversation
GET	`/api/conversations/:id`	Get specific conversation
DELETE	`/api/conversations/:id`	Delete conversation
POST	`/api/conversations/:id/messages`	Send message

Table 4.3: Conversation API Endpoints

4.3 Request/Response Examples

4.3.1 Device Code Request

`POST /api/auth/device/code`

Request:

```
POST /api/auth/device/code
Content-Type: application/json

{}
```

Response:

```
{
  "deviceCode": "550e8400-e29b-41d4-a716-446655440000",
  "userCode": "ABCD-1234",
  "verificationUri": "http://localhost:3000/device",
  "expiresIn": 600,
  "interval": 5
}
```

4.3.2 Session Retrieval

GET /api/me

Request:

```
GET /api/me
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Response:

```
{
  "user": {
    "id": "usr_123",
    "name": "John Doe",
    "email": "john@example.com",
    "image": "https://..."
  },
  "session": {
    "id": "ses_456",
    "token": "...",
    "expiresAt": "2026-02-05T12:00:00Z"
  }
}
```

4.4 Middleware Stack

Express Middleware

cors()	CORS configuration for client origins
express.json()	Parse JSON request bodies
express.urlencoded()	Parse URL-encoded bodies
morgan()	HTTP request logging
helmet()	Security headers
authMiddleware	JWT token validation
errorHandler	Centralized error handling

4.4.1 Authentication Middleware

Auth Middleware Implementation

```
// middleware/auth.js
export async function authMiddleware(req, res, next) {
  const token = req.headers.authorization?.replace('Bearer ', '')

  if (!token) {
    return res.status(401).json({ error: 'No token provided' })
  }

  try {
    const session = await prisma.session.findUnique({
      where: { token },
      include: { user: true },
    })

    if (!session || session.expiresAt < new Date()) {
      return res.status(401).json({ error: 'Invalid or expired token' })
    }

    req.user = session.user
    req.session = session
    next()
  } catch (error) {
    res.status(500).json({ error: 'Authentication failed' })
  }
}
```

4.5 AI Service Integration

4.5.1 Google AI SDK Configuration

AI Service Setup

```
// services/ai.js
import { google } from '@ai-sdk/google'
import { generateText, streamText } from 'ai'

const model = google('gemini-1.5-pro-latest')

export async function generateAIResponse(messages, mode = 'chat') {
  const result = await generateText({
    model,
    messages,
    temperature: mode === 'agent' ? 0.7 : 0.5,
    maxTokens: 2048,
  })

  return result.text
}

export async function streamAIResponse(messages) {
  const stream = await streamText({
    model,
    messages,
  })

  return stream.toTextStream()
}
```

4.6 Error Handling

Error Response Format

All API errors follow a consistent format:

```
{
  "error": true,
  "message": "Human-readable error message",
  "code": "ERROR_CODE",
  "details": {
    "field": "specific error details"
  }
}
```

Common error codes:

- `AUTH_REQUIRED` - Missing authentication
- `INVALID_TOKEN` - Token expired or invalid
- `NOT_FOUND` - Resource not found
- `VALIDATION_ERROR` - Invalid input data
- `RATE_LIMIT` - Too many requests

4.7 Database Services

Prisma Service Layer

```
// services/conversation.js
export class ConversationService {
  async createConversation(userId, data) {
    return await prisma.conversation.create({
      data: {
        userId,
        title: data.title,
        mode: data.mode,
      },
    })
  }

  async addMessage(conversationId, role, content) {
    return await prisma.message.create({
      data: {
        conversationId,
        role,
        content,
      },
    })
  }

  async getConversationHistory(conversationId) {
    return await prisma.message.findMany({
      where: { conversationId },
      orderBy: { createdAt: 'asc' },
    })
  }
}
```

Chapter 5

Orbit CLI

The Orbit CLI provides a powerful command-line interface for interacting with Neuro-CLI's AI capabilities directly from the terminal. This chapter covers CLI usage, commands, and development.

5.1 CLI Architecture

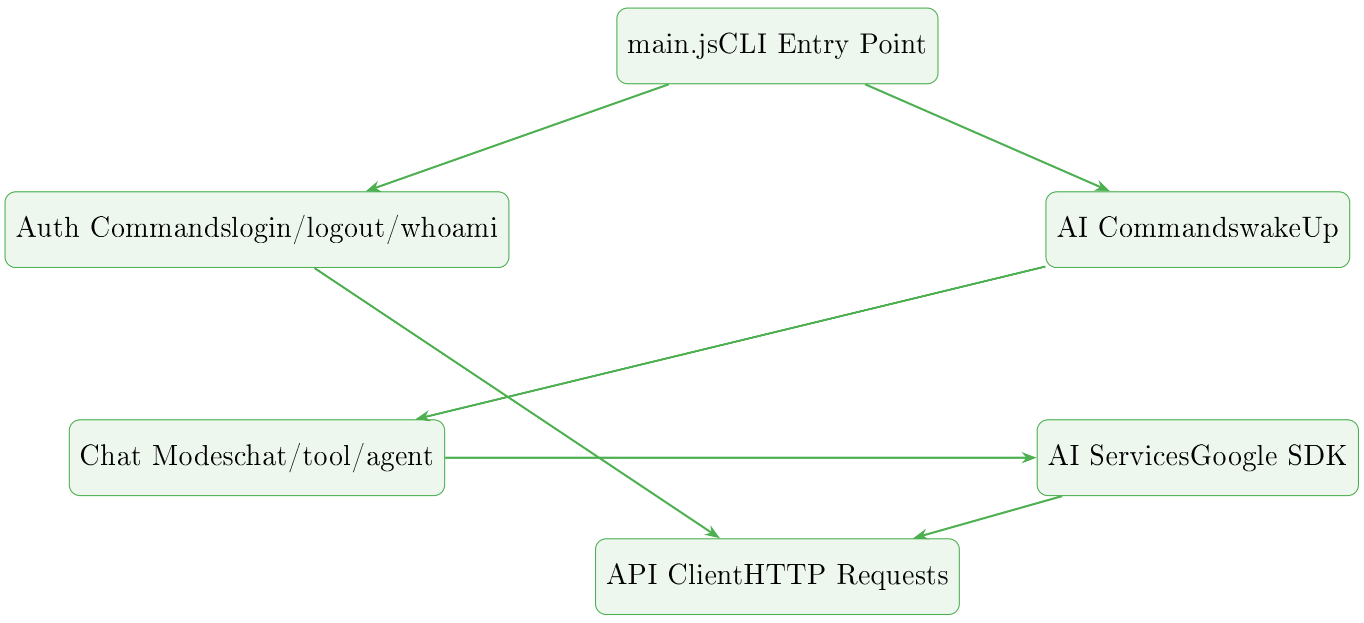

Figure 5.1: CLI Command Structure

5.2 Available Commands

5.2.1 Authentication Commands

Command	Usage	Description
`login`	`npm run cli - login`	Authenticate via device flow
`logout`	`npm run cli - logout`	End current session
`whoami`	`npm run cli - whoami`	Display user information

Table 5.1: Authentication Commands

5.2.2 AI Interaction Commands

Command	Usage	Description
`wakeUp`	`npm run cli - wakeUp`	Start AI chat session

Table 5.2: AI Commands

5.3 Command Usage Examples

5.3.1 Login Flow

Authentication Workflow

```
$ npm run cli -- login


     _   _                       ____ _     ___
    | \ | | ___ _   _ _ __ ___  / ___| |   |_ _|
    |  \| |/ _ \ | | | ’__/ _ \| |   | |    | |
    | |\  |  __/ |_| | | | (_) | |___| |___ | |
    |_| \_|\___|\__,_|_|  \___/ \____|_____|___|

      Initiating device flow authentication...

      Your user code: ABCD-1234

      Visit: http://localhost:3000/device
      Enter the code above to authenticate

      Waiting for authorization...

      Authentication successful!
      Welcome, John Doe!

      Session expires: 2026-02-05 12:00:00
```

5.3.2 Chat Session

AI Chat Interaction

```
$ npm run cli -- wakeUp

      Select chat mode:
       chat   - Free-form conversation
       tool   - Function-calling mode
       agent  - Autonomous agent


      Starting agent mode...


     Neuro-CLI Agent Mode
     Type 'exit' to quit, 'clear' to reset


    You: What's the weather like today?

    Agent: I apologize, but I don't have access to
    real-time weather data. To get current weather
    information, I would need:

    1. Your location
    2. Access to a weather API

    Would you like me to help you set up a weather
    tool for future queries?

    You: exit

      Conversation saved.
      ID: conv_789
      Thank you for using Neuro-CLI!
```

5.4 CLI Features

5.4.1 Terminal UI Elements

UI Components

ASCII Art	Branded headers with Figlet
Colored Output	Syntax highlighting with Chalk
Spinners	Loading animations with Ora
Prompts	Interactive inputs with Clack/Inquirer
Boxes	Framed messages with Boxen
Progress Bars	Task progress indicators
Markdown	Formatted AI responses

5.4.2 Interactive Prompts

Prompt Examples

```
// Using Clack for beautiful prompts
import * as clack from ’@clack/prompts’

const mode = await clack.select({
  message: ’Select chat mode:’,
  options: [
    { value: ’chat’, label: ’Chat - Free-form conversation’ },
    { value: ’tool’, label: ’Tool - Function-calling mode’ },
    { value: ’agent’, label: ’Agent - Autonomous agent’ },
  ],
})

const name = await clack.text({
  message: ’What is your name?’,
  placeholder: ’Anonymous’,
  validate: (value) => {
    if (!value) return ’Name is required’
  },
})
```

5.5 Configuration

5.5.1 CLI Config File

/.neurocli/config.json

```
{
  "apiUrl": "http://localhost:3005",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "userId": "usr_123",
  "preferences": {
    "theme": "dark",
    "language": "en",
    "markdown": true
  },
  "lastUsed": "2026-01-29T10:00:00Z"
}
```

5.5.2 Token Storage

Secure Token Management

Tokens are stored in the user's home directory:

- **Location:** `~/.neurocli/`
- **Permissions:** 600 (owner read/write only)
- **Encryption:** Tokens stored in plaintext (file permissions provide security)
- **Expiration:** Automatically handled by server

5.6 Chat Modes

5.6.1 Mode Comparison

Mode	Capabilities	Use Cases
Chat	Basic conversation, no tools	General questions, brainstorming, explanations
Tool	Function calling, tool use	Calculations, API calls, data processing
Agent	Autonomous actions, multi-step planning	Complex tasks, research, automated workflows

Table 5.3: AI Chat Modes

5.6.2 Chat Mode Features

Chat Mode

- Simple conversational interface
- No function calling overhead
- Fast response times
- Best for Q&A and casual interaction

Tool Mode

- AI can invoke predefined functions
- Requires tool definitions in code
- Useful for calculations and API integration
- Example: weather lookup, math operations

Agent Mode

- Autonomous decision-making
- Can plan multi-step tasks
- Higher token usage
- Requires careful prompt engineering

5.7 Development Guide

5.7.1 Adding New Commands

Command Template

```
// commands/mycommand.js
import { Command } from 'commander'
import * as clack from '@clack/prompts'

export function registerMyCommand(program) {
  program
    .command('mycommand')
    .description('Description of my command')
    .option('-f, --flag', 'Example flag')
    .action(async (options) => {
      clack.intro('My Command')

      // Command logic here

      clack.outro('Done!')
    })
}
```

5.7.2 Testing CLI Commands

Testing Workflow

```
# Run specific command
npm run cli -- mycommand

# Run with debug output
DEBUG=* npm run cli -- mycommand

# Test with different options
npm run cli -- mycommand --flag
```

Chapter 6

Database Schema

This chapter details the PostgreSQL database schema managed by Prisma ORM, including table structures, relationships, and data management strategies.

6.1 Schema Overview

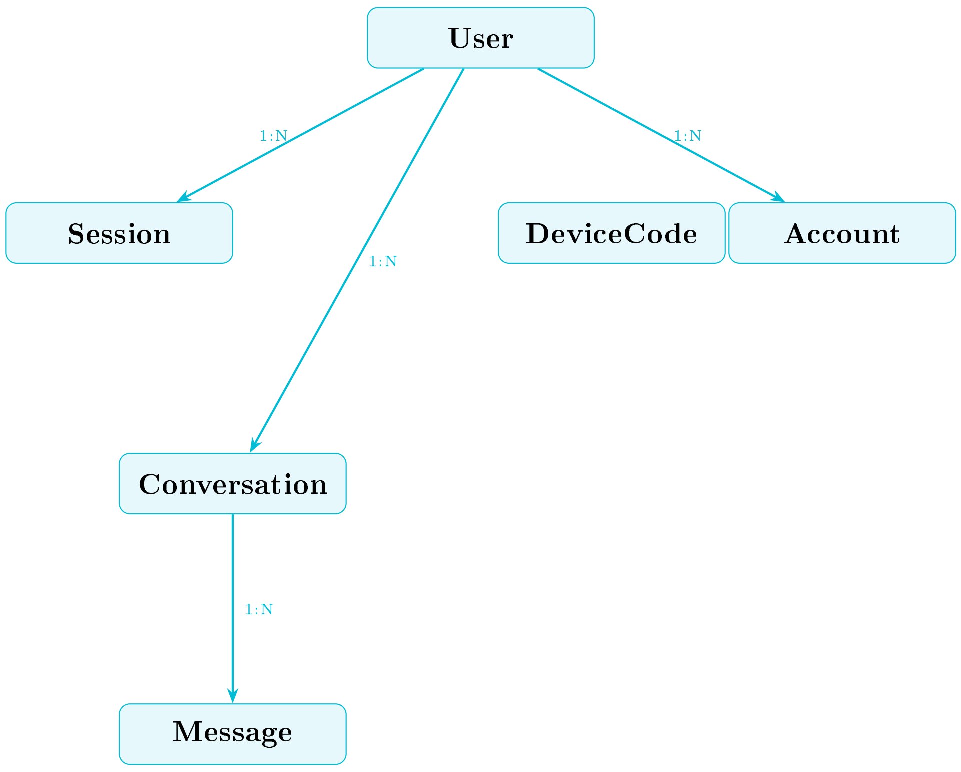

Figure 6.1: Database Relationships

6.2 Core Tables

6.2.1 User Table

User Schema

```
model User {
  id            String        @id @default(cuid())
  name          String?
  email         String        @unique
  emailVerified Boolean       @default(false)
  image         String?
  createdAt     DateTime      @default(now())
  updatedAt     DateTime      @updatedAt

  sessions      Session[]
  accounts      Account[]
  conversations Conversation[]
}
```

Purpose: Store user account information
Key Fields:

- `id`: Unique identifier (CUID)
- `email`: Unique email address
- `emailVerified`: Verification status

6.2.2 Session Table

Session Schema

```
model Session {
  id           String   @id @default(cuid())
  token        String   @unique
  expiresAt    DateTime
  userId       String
  ipAddress    String?
  userAgent    String?
  createdAt    DateTime @default(now())

  user         User     @relation(fields: [userId],
                                  references: [id],
                                  onDelete: Cascade)
}
```

Purpose: Manage active user sessions

Key Features:

- Token-based authentication
- Expiration tracking
- IP and user agent logging
- Cascade delete with user

6.2.3 Conversation Table

Conversation Schema

```
model Conversation {
  id        String    @id @default(cuid())
  userId    String
  title     String
  mode      String    // 'chat', 'tool', or 'agent'
  createdAt DateTime  @default(now())
  updatedAt DateTime  @updatedAt

  user      User      @relation(fields: [userId],
                                references: [id],
                                onDelete: Cascade)
  messages  Message[]

  @@index([userId, createdAt])
}
```

Purpose: Group related messages into conversations
Indexed Fields: userId + createdAt for efficient queries

6.2.4 Message Table

Message Schema

```
model Message {
  id              String       @id @default(cuid())
  conversationId  String
  role            String       // 'user' or 'assistant'
  content         String       @db.Text
  createdAt       DateTime     @default(now())

  conversation    Conversation @relation(fields: [conversationId],
                                         references: [id],
                                         onDelete: Cascade)

  @@index([conversationId, createdAt])
}
```

Purpose: Store individual messages in conversations
Note: Uses `@db.Text` for large content storage

6.2.5 DeviceCode Table

DeviceCode Schema

```
model DeviceCode {
  id         String   @id @default(cuid())
  deviceCode String   @unique
  userCode   String   @unique
  userId     String?
  expiresAt  DateTime
  status     String   @default("pending")
  createdAt  DateTime @default(now())

  @@index([userCode])
  @@index([deviceCode])
}
```

Purpose: Manage device flow authentication
Status Values:

- `pending`: Awaiting user approval
- `approved`: User approved device
- `expired`: Code has expired

6.3 Database Operations

6.3.1 Prisma Client Usage

Common Queries

```
import { PrismaClient } from '@prisma/client'
const prisma = new PrismaClient()

// Create user
const user = await prisma.user.create({
  data: {
    email: 'user@example.com',
    name: 'John Doe',
  },
})

// Find user with sessions
const userWithSessions = await prisma.user.findUnique({
  where: { email: 'user@example.com' },
  include: { sessions: true },
})

// Create conversation with first message
const conversation = await prisma.conversation.create({
  data: {
    userId: user.id,
    title: 'New Chat',
    mode: 'chat',
    messages: {
      create: {
        role: 'user',
        content: 'Hello!',
      },
    },
  },
})

// Get conversation history
const messages = await prisma.message.findMany({
  where: { conversationId: conversation.id },
  orderBy: { createdAt: 'asc' },
})
```

6.3.2 Migrations

Migration Commands

```
# Create migration
npx prisma migrate dev --name add_user_preferences

# Apply migrations to production
npx prisma migrate deploy

# Reset database (development only)
npx prisma migrate reset

# View migration status
npx prisma migrate status
```

6.4 Data Retention

Cleanup Policies

Expired Sessions	Deleted automatically after expiration
Device Codes	Expired codes cleaned up after 24 hours
Conversations	Retained indefinitely unless user deletes
Messages	Cascade deleted with parent conversation

6.4.1 Cleanup Script

Automated Cleanup

```
// scripts/cleanup.js
async function cleanupExpiredData() {
  // Delete expired sessions
  await prisma.session.deleteMany({
    where: {
      expiresAt: { lt: new Date() },
    },
  })

  // Delete old device codes
  const oneDayAgo = new Date(Date.now() - 24 * 60 * 60 * 1000)
  await prisma.deviceCode.deleteMany({
    where: {
      createdAt: { lt: oneDayAgo },
      status: { in: ['expired', 'approved'] },
    },
  })
}

// Run daily via cron
```

6.5 Indexing Strategy

Table	**Index**	**Purpose**
User	email (unique)	Fast user lookup by email
Session	token (unique)	Quick session validation
Conversation	userId + createdAt	Efficient user conversation listing
Message	conversationId + createdAt	Fast message retrieval
DeviceCode	userCode, deviceCode	Device flow lookups

Table 6.1: Database Indexes

Chapter 7

Deployment & Production

This chapter covers deployment strategies, production configuration, monitoring, and maintenance procedures for Neuro-CLI.

7.1 Production Checklist

Pre-Deployment Checklist

Security:

Change all default secrets and API keys

Enable HTTPS with valid SSL certificates

Configure CORS for production domains

Set secure cookie attributes

Enable rate limiting

Review database access permissions

Configuration:

Set `NODE_ENV=production`

Configure production database

Set up environment variables

Configure logging

Set up monitoring

Testing:

Run full test suite

Perform load testing

Test backup/restore procedures

Verify all API endpoints

7.2 Deployment Options

7.2.1 Option 1: Traditional VPS

VPS Deployment Steps

1. Server Setup

```
# Update system
sudo apt update && sudo apt upgrade -y

# Install Node.js
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt install -y nodejs

# Install PostgreSQL
sudo apt install -y postgresql postgresql-contrib

# Install PM2
sudo npm install -g pm2
```

2. Application Deployment

```
# Clone repository
git clone <repo-url> /var/www/neurocli
cd /var/www/neurocli

# Install dependencies
cd server && npm install --production
cd ../client && npm install --production

# Build client
npm run build

# Start with PM2
cd ../server
pm2 start ecosystem.config.js
```

7.2.2 PM2 Configuration

ecosystem.config.js

```
module.exports = {
  apps: [
    {
      name: 'neurocli-api',
      script: './src/index.js',
      instances: 2,
      exec_mode: 'cluster',
      env: {
        NODE_ENV: 'production',
        PORT: 3005,
      },
      error_file: './logs/err.log',
      out_file: './logs/out.log',
      log_date_format: 'YYYY-MM-DD HH:mm:ss Z',
    },
  ],
}
```

7.2.3 Option 2: Docker Deployment

Dockerfile - Server

```
FROM node:20-alpine

WORKDIR /app

COPY package*.json ./
RUN npm install --production

COPY . .

RUN npx prisma generate

EXPOSE 3005

CMD ["npm", "start"]
```

docker-compose.yml

```yaml
version: '3.8'

services:
  api:
    build: ./server
    ports:
      - "3005:3005"
    environment:
      DATABASE_URL: postgresql://user:pass@db:5432/neurocli
      NODE_ENV: production
    depends_on:
      - db

  client:
    build: ./client
    ports:
      - "3000:3000"
    environment:
      NEXT_PUBLIC_API_URL: http://api:3005

  db:
    image: postgres:15-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: neurocli
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass

volumes:
  postgres_data:
```

7.3 Nginx Configuration

Nginx Reverse Proxy

```
# /etc/nginx/sites-available/neurocli
server {
    listen 80;
    server_name neurocli.example.com;
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl http2;
    server_name neurocli.example.com;

    ssl_certificate /etc/letsencrypt/live/neurocli.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/neurocli.example.com/privkey.pem;

    # API
    location /api/ {
        proxy_pass http://localhost:3005;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }

    # Web App
    location / {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
    }
}
```

7.4 Monitoring & Logging

7.4.1 Application Monitoring

PM2 Monitoring

```
# View running processes
pm2 list

# Monitor resources
pm2 monit

# View logs
pm2 logs neurocli-api

# Save PM2 configuration
pm2 save

# Set PM2 to start on boot
pm2 startup
```

7.4.2 Database Backups

Backup Script

```
#!/bin/bash
# backup.sh

BACKUP_DIR="/var/backups/neurocli"
DATE=$(date +%Y%m%d_%H%M%S)
FILENAME="neurocli_$DATE.sql"

# Create backup directory
mkdir -p $BACKUP_DIR

# Dump database
pg_dump neurocli > "$BACKUP_DIR/$FILENAME"

# Compress
gzip "$BACKUP_DIR/$FILENAME"

# Delete backups older than 30 days
find $BACKUP_DIR -name "*.sql.gz" -mtime +30 -delete

echo "Backup completed: $FILENAME.gz"
```

Schedule with Cron:

```
# Daily at 2 AM
0 2 * * * /path/to/backup.sh
```

7.5 Maintenance

7.5.1 Update Procedure

Safe Update Process

```
# 1. Backup database
./backup.sh

# 2. Pull latest code
git pull origin main

# 3. Update dependencies
npm install

# 4. Run migrations
npx prisma migrate deploy

# 5. Rebuild client
cd client && npm run build

# 6. Restart services
pm2 restart all

# 7. Verify
curl https://neurocli.example.com/api/health
```

7.5.2 Health Check Endpoint

Health Check Implementation

```
// routes/health.js
app.get('/api/health', async (req, res) => {
  const health = {
    status: 'ok',
    timestamp: new Date().toISOString(),
    uptime: process.uptime(),
    database: 'unknown',
  }

  try {
    await prisma.$queryRaw`SELECT 1`
    health.database = 'connected'
  } catch (error) {
    health.database = 'error'
    health.status = 'degraded'
  }

  const statusCode = health.status === 'ok' ? 200 : 503
  res.status(statusCode).json(health)
})
```

7.6 Troubleshooting

Common Production Issues

High Memory Usage:

- Check for memory leaks with `pm2 monit`
- Restart application: `pm2 restart all`
- Review database query efficiency

Database Connection Errors:

- Verify PostgreSQL is running
- Check connection pool settings
- Review `DATABASE_URL` configuration

Slow Response Times:

- Enable query logging in Prisma
- Add database indexes
- Implement caching layer
- Scale horizontally with PM2 clusters

Appendix A

Quick Reference

A.1 Environment Variables

Variable	**Required**	**Description**
DATABASE_URL	Yes	PostgreSQL connection string
BETTER_AUTH_SECRET	Yes	32-char authentication secret
GOOGLE_AI_API_KEY	Yes	Google AI SDK API key
PORT	No	API server port (default: 3005)
NODE_ENV	No	Environment (development/production)
CORS_ORIGINS	No	Allowed CORS origins

A.2 CLI Commands

```
# Authentication
npm run cli -- login          # Device flow login
npm run cli -- logout         # End session
npm run cli -- whoami         # Show current user

# AI Interaction
npm run cli -- wakeUp         # Start chat session

# Server Management
npm run dev                   # Start dev server
npm run build                 # Build for production
npm start                     # Start production server

# Database
npx prisma generate           # Generate Prisma Client
npx prisma db push            # Sync schema to database
npx prisma studio             # Open database GUI
npx prisma migrate dev        # Create migration
```

A.3 API Quick Reference

```
# Authentication
POST   /api/auth/login
POST   /api/auth/device/code
POST   /api/auth/device/token

# Session
GET    /api/me
GET    /api/sessions
DELETE /api/sessions/:id

# Conversations
GET    /api/conversations
POST   /api/conversations
POST   /api/conversations/:id/messages
```

A.4 Support & Resources

Getting Help

- **Author**: Mausam Kar
- **Portfolio**: `https://mausam04.vercel.app`
- **License**: ISC License
- **Built with**: Next.js, Express, Google AI SDK

Thank you for using Neuro-CLI (Orbit)!

Made with love by Mausam Kar