Complete AI Web Development Setup Guide

A comprehensive reference for building web applications with AI integration

---

Table of Contents

VS Code AI Assistant Configuration

File: `new-agent.yaml` or `.vscode/ai-config.yaml`

yaml

```yaml
# Free AI Agent Configuration for VS Code
# This configuration uses free/open-source AI models
name: Free AI Assistant
version: 1.0.0
schema: v1

# System prompt to optimize the agent's behavior
systemPrompt: |
  You are an expert programming assistant integrated into VS Code, specializing in full-stack web development.

  ## Core Responsibilities:
  - Code writing, debugging, refactoring, and code explanation
  - Provide concise yet thorough responses
  - Always consider best practices, security implications, and performance

  ## Default Tech Stack:

  **Frontend (unless specified otherwise):**
  - HTML5
  - SCSS/CSS
  - Vanilla JavaScript (or specify framework if needed)
  - Responsive design for mobile/tablet/desktop
  - React available but may not always be used

  **Backend:**
  - Python with Flask framework
  - SQLAlchemy for database ORM
  - Flask-Migrate for database migrations
  - RESTful API design

  **Database Management:**
  - SQLite for local development and prototyping
  - PostgreSQL for production (Neon - 10GB free tier)
  - MongoDB Atlas (512MB free tier) - alternative for NoSQL
  - MySQL via PlanetScale (5GB free tier) - alternative

  **AI Integration:**

  *Development (Local):*
  - Ollama for local AI (free, unlimited, private)
  - Models: llama3.1, qwen2.5-coder, deepseek-coder
  - Runs on localhost:11434
```

Production (Deployed):
- Groq API (FREE tier - primary choice)
  * Model: llama-3.1-8b-instant
  * Limits: 30 requests/min, 14,400 requests/day
  * Cost: $0 for free tier
- OpenAI API (Paid backup)
  * Model: gpt-4o-mini
  * Cost: ~$0.15 per million tokens

Security:
- NEVER expose API keys in frontend code
- All AI API calls through backend only
- Use environment variables for sensitive data
- Implement rate limiting (400-500 requests/day per user)

Deployment & Hosting:

For Static Landing Pages (FREE):
- GitHub Pages, Netlify, or Vercel
- No backend needed
- Forms: Formspree, Netlify Forms

For Full-Stack Apps:
- Render (free tier) or Railway ($5 credit)
- Neon PostgreSQL (10GB free)
- Docker for containerization (optional)

For Chat UI with AI:
- Frontend: GitHub Pages/Netlify
- Backend: Flask on Render
- AI: Groq (free) or OpenAI
- Database: Neon PostgreSQL (optional for history)
- Rate Limiting: 400-500 requests/day per user

DevOps:
- Git for version control
- GitHub for repository management
- GitHub Actions for CI/CD
- Docker for containerization (when needed)
- Kubernetes only for large-scale production (usually overkill)

Workflow:
1. **Always ask for confirmation** before starting a new task
2. Clarify requirements and scope before coding
3. Explain your approach for complex solutions

```
4. Provide code with appropriate comments
5. Suggest testing strategies when relevant
6. Prioritize free/low-cost solutions

## Best Practices:
- Write clean, maintainable, well-documented code
- Follow security best practices (input validation, secure auth)
- Implement error handling and logging
- Use environment variables for sensitive data
- Never commit secrets to Git
- Consider scalability and performance
- Start simple, scale when needed
- Monitor usage and costs in production

## Rate Limiting:
- Per-user limit: 400-500 requests per day
- Implement in backend middleware
- Clear error messages when limits exceeded
- Exponential backoff for API retries

# Models define which AI models this agent can use
# These are all free options you can use
models:
  # Option 1: Local models via Ollama (completely free, runs on your machine)
  - name: llama-3.2
    uses: ollama/llama3.2:3b
    title: Llama 3.2 (Fast, Local)

  - name: qwen-coder
    uses: ollama/qwen2.5-coder:7b
    title: Qwen Coder (Best for coding)

  - name: deepseek-coder
    uses: ollama/deepseek-coder-v2:16b
    title: DeepSeek Coder (Advanced coding)

  - name: codellama
    uses: ollama/codellama:13b
    title: Code Llama (Meta's coding model)

  # Option 2: Free API-based models
  - name: groq-llama
    provider: groq
    model: llama-3.1-70b-versatile
    apiKey: ${GROQ_API_KEY}
    title: Groq Llama (Fast cloud API)
```

```
    title: Groq Llama (Fast cloud API)

  - name: together-ai
    provider: together
    model: meta-llama/Llama-3.2-11B-Vision-Instruct-Turbo
    apiKey: ${TOGETHER_API_KEY}
    title: Together AI (Free tier available)

  - name: openrouter-free
    provider: openrouter
    model: google/gemini-flash-1.5-8b
    apiKey: ${OPENROUTER_API_KEY}
    title: OpenRouter (Very cheap usage)

# Context providers (what the agent can access)
contextProviders:
  - name: codebase
  - name: open-file
  - name: terminal

# Slash commands for quick actions
slashCommands:
  - name: explain
    description: Explain the selected code

  - name: fix
    description: Fix issues in the selected code

  - name: test
    description: Generate tests for the selected code

  - name: refactor
    description: Refactor the selected code

# MCP Servers (optional tools)
mcpServers: []
```

System Prompt (Complete)

This is the detailed system prompt that guides your AI assistant:

Core Responsibilities

- Code writing, debugging, refactoring, and code explanation
- Provide concise yet thorough responses
- Always consider best practices, security implications, and performance

Tech Stack Details

Frontend

- HTML5
- SCSS/CSS
- Vanilla JavaScript (or specify framework if needed)
- Responsive design for mobile/tablet/desktop
- React available but may not always be used

Backend

- Python with Flask framework
- SQLAlchemy for database ORM
- Flask-Migrate for database migrations
- RESTful API design
- Async support with asyncio where beneficial

Database Management

- **SQLite:** Local development and prototyping
- **PostgreSQL:** Production (Neon - 10GB free tier recommended)
- **MongoDB Atlas:** 512MB free tier - alternative for NoSQL
- **MySQL:** PlanetScale (5GB free tier) - alternative

Tech Stack Overview

Complete Technology Breakdown

Layer	Development	Production
Frontend	HTML/SCSS/JS	Same (static hosting)
Backend	Flask (localhost)	Flask on Render/Railway
Database	SQLite	PostgreSQL (Neon) / MongoDB
AI	Ollama (local)	Groq (free) / OpenAI (paid)
Hosting	localhost:5000	Render/Railway/Netlify
Version Control	Git	GitHub
CI/CD	Manual	GitHub Actions

Free Hosting Options

Static Sites (Landing Pages)

Platform	Free Tier	Best For	Limitations
GitHub Pages	Unlimited	Simple static sites	No backend, no server-side code
Netlify	100GB bandwidth/mo	Static + forms	100 form submissions/mo free
Vercel	Unlimited	Next.js, static sites	Serverless functions limited
Cloudflare Pages	Unlimited	Static sites	Learning curve

Recommended: GitHub Pages for pure static, Netlify if you need forms

Full-Stack Apps (with Backend)

Platform	Free Tier	Database	Best For
Render	750 hours/mo	PostgreSQL (90 days)	Flask apps, auto-deploy
Railway	$5 credit/mo	Included	Small projects, excellent DX
Fly.io	3 VMs free	Extra cost	Python apps, global edge
PythonAnywhere	Limited CPU	MySQL included	Python-specific, beginner-friendly

Recommended: Render for production-ready free tier, Railway for best experience

When to Use What

Use GitHub Pages when:

- Just a landing page/portfolio
- No backend needed
- No user data to store
- Pure HTML/CSS/JS

Use Netlify when:

- Static site + contact forms
- Need some serverless functions
- Want easy deployment

Use Render when:

- Need Flask backend
- Need database
- Building chat UI with AI
- Want free production hosting

Use Railway when:

- Small project
- Want best developer experience
- $5/month budget is acceptable

Free Database Options

Relational Databases

Database	Provider	Free Tier	Best For
SQLite	Local file	Unlimited	Development, prototypes
PostgreSQL	Neon	10GB	Production apps (recommended)
PostgreSQL	Supabase	500MB	Apps needing auth/realtime
PostgreSQL	ElephantSQL	20MB	Tiny projects
PostgreSQL	Render	90 days	Testing (expires)
MySQL	PlanetScale	5GB	Production MySQL needs
MySQL	PythonAnywhere	Included	Simple projects

NoSQL Databases

Database	Provider	Free Tier	Best For
MongoDB	Atlas	512MB	Document storage, NoSQL
Firebase	Google	1GB + 50K reads/day	Real-time apps, mobile backends
Redis	Upstash	10K commands/day	Caching, sessions

Recommendations by Use Case

For Chat UI with AI:

- **Primary:** Neon PostgreSQL (10GB free, serverless, excellent)
- **Alternative:** MongoDB Atlas (512MB, good for JSON-heavy data)

For Landing Pages:

- **None needed** - use Formspree for forms

For E-commerce:

- **PostgreSQL** via Neon (structured data, transactions)

For Real-time Apps:

- **Firebase Firestore** (real-time sync)

AI Integration (Local vs Production)

Development (Local)

Use Ollama - completely free, unlimited, private

```bash
# Install Ollama
curl -fsSL https://ollama.com/install.sh | sh

# Pull models
ollama pull llama3.1:latest
ollama pull qwen2.5-coder:7b
ollama pull deepseek-coder-v2:16b

# Run Ollama server
ollama serve
```

Advantages:

- ✅Free and unlimited
- ✅Private (data never leaves your machine)
- ✅No API keys needed
- ✅Works offline

Disadvantages:

- ❌Requires 4-8GB RAM
- ❌Slower than cloud APIs
- ❌Only works on your machine

Production (Deployed)

Option 1: Groq (FREE - Recommended)

Specs:

- **Cost:** FREE
- **Model:** llama-3.1-8b-instant
- **Limits:** 30 req/min, 14,400 req/day
- **Speed:** Very fast (fastest available)

Sign up: https://console.groq.com/

```python
from groq import AsyncGroq

client = AsyncGroq(api_key=os.getenv('GROQ_API_KEY'))

response = await client.chat.completions.create(
    model="llama-3.1-8b-instant",
    messages=[{"role": "user", "content": "Hello!"}]
)
```

When to use:

- Small to medium projects
- Budget is tight
- Speed is important
- 14,400 requests/day is enough

Option 2: OpenAI (Paid)

Specs:

- **Cost:** ~$0.15 per million tokens
- **Model:** gpt-4o-mini (cheapest)
- **Limits:** None (pay-per-use)
- **Quality:** Better reasoning

Sign up: https://platform.openai.com/

```python
from openai import AsyncOpenAI

client = AsyncOpenAI(api_key=os.getenv('OPENAI_API_KEY'))

response = await client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "user", "content": "Hello!"}]
)
```

When to use:

- Need better quality
- Exceed Groq limits
- Budget allows $5-20/month

Comparison Table

Feature	Local (Ollama)	Groq (Free)	OpenAI (Paid)
Cost	$0	$0	~$5-20/mo
Speed	Slow-Medium	Very Fast	Fast
Quality	Good	Good	Best
Privacy	100% Private	Cloud	Cloud
Limits	None	14.4K/day	Pay-per-use
Offline	Yes	No	No
Setup	Complex	Easy	Easy

Python AI Client Code

File: `services/ai_client.py`

python

```python
from typing import Optional
import logging
import httpx
import os
from datetime import datetime, timedelta
from openai import AsyncOpenAI
from groq import AsyncGroq, RateLimitError
import asyncio

logger = logging.getLogger(__name__)

# Configuration
ENVIRONMENT = os.getenv('ENVIRONMENT', 'development')
OLLAMA_BASE_URL = os.getenv('OLLAMA_BASE_URL', "http://localhost:11434")
OLLAMA_MODEL = os.getenv('OLLAMA_MODEL', "llama3.1:latest")

# Rate limiting configuration
USER_DAILY_LIMIT = int(os.getenv('USER_DAILY_LIMIT', 500))


class RateLimiter:
    """Simple in-memory rate limiter"""

    def __init__(self, limit_per_day: int = USER_DAILY_LIMIT):
        self.limit = limit_per_day
        self.usage = {}

    def check_limit(self, user_id: str) -> tuple[bool, int]:
        """Check if user is within rate limit"""
        now = datetime.now()

        if user_id not in self.usage:
            self.usage[user_id] = {
                'count': 0,
                'reset_time': now + timedelta(days=1)
            }

        user_data = self.usage[user_id]

        if now >= user_data['reset_time']:
            user_data['count'] = 0
            user_data['reset_time'] = now + timedelta(days=1)
```

```
        if user_data['count'] >= self.limit:
            return False, 0

        user_data['count'] += 1
        remaining = self.limit - user_data['count']
        return True, remaining

    def get_remaining(self, user_id: str) -> int:
        """Get remaining requests for user"""
        if user_id not in self.usage:
            return self.limit

        user_data = self.usage[user_id]
        now = datetime.now()

        if now >= user_data['reset_time']:
            return self.limit

        return max(0, self.limit - user_data['count'])


class OllamaClient:
    """Local Ollama client for development"""

    def __init__(self, base_url: str = OLLAMA_BASE_URL, model: str = OLLAMA_MODEL):
        self.base_url = base_url
        self.model = model

    async def chat(self, message: str, context: str = "") -> str:
        try:
            async with httpx.AsyncClient(timeout=30.0) as client:
                prompt = f"{context}\n\nUser: {message}\nAssistant:" if context else message
                response = await client.post(
                    f"{self.base_url}/api/generate",
                    json={
                        "model": self.model,
                        "prompt": prompt,
                        "stream": False,
                        "options": {
                            "temperature": 0.7,
                            "num_predict": 500
                        }
                    }
                )
```

```
            if response.status_code == 200:
                result = response.json()
                return result.get("response", "No response from AI")
            else:
                logger.error(f"Ollama error: {response.status_code}")
                return "AI service temporarily unavailable"

        except httpx.ConnectError:
            logger.error("Cannot connect to Ollama")
            return "AI assistant is offline. Please start Ollama with 'ollama serve'"
        except Exception as e:
            logger.error(f"Error calling Ollama: {e}")
            return "I'm having trouble processing your request right now."


class GroqClient:
    """Groq cloud API client (FREE tier)"""

    def __init__(self):
        api_key = os.getenv('GROQ_API_KEY')
        if not api_key:
            raise ValueError("GROQ_API_KEY not set")

        self.client = AsyncGroq(api_key=api_key)
        self.model = "llama-3.1-8b-instant"

    async def chat(self, message: str, context: str = "", max_retries: int = 3) -> str:
        for attempt in range(max_retries):
            try:
                messages = []
                if context:
                    messages.append({"role": "system", "content": context})
                messages.append({"role": "user", "content": message})

                response = await self.client.chat.completions.create(
                    model=self.model,
                    messages=messages,
                    max_tokens=500,
                    temperature=0.7
                )

                return response.choices[0].message.content

            except RateLimitError as e:
                if attempt < max_retries - 1:
                    wait_time = 60
```

```
                wait_time = 60
                logger.warning(f"Groq rate limited, waiting {wait_time}s")
                await asyncio.sleep(wait_time)
            else:
                return "I'm receiving too many requests. Please try again in a minute."

        except Exception as e:
            logger.error(f"Groq error: {e}")
            if attempt < max_retries - 1:
                await asyncio.sleep(2 ** attempt)
            else:
                return "I'm having trouble processing your request."

    return "Service temporarily unavailable."


class OpenAIClient:
    """OpenAI API client (paid backup)"""

    def __init__(self):
        api_key = os.getenv('OPENAI_API_KEY')
        if not api_key:
            raise ValueError("OPENAI_API_KEY not set")

        self.client = AsyncOpenAI(api_key=api_key)
        self.model = "gpt-4o-mini"

    async def chat(self, message: str, context: str = "") -> str:
        try:
            messages = []
            if context:
                messages.append({"role": "system", "content": context})
            messages.append({"role": "user", "content": message})

            response = await self.client.chat.completions.create(
                model=self.model,
                messages=messages,
                max_tokens=500,
                temperature=0.7
            )

            return response.choices[0].message.content

        except Exception as e:
            logger.error(f"OpenAI error: {e}")
            return "I'm having trouble processing your request."
```

```
        return "I'm having trouble processing your request."


# Global instances
ai_client = None
rate_limiter = RateLimiter(USER_DAILY_LIMIT)


def init_ai():
    """Initialize AI client based on environment"""
    global ai_client

    try:
        if ENVIRONMENT == 'production':
            provider = os.getenv('AI_PROVIDER', 'groq')

            if provider == 'groq':
                ai_client = GroqClient()
                logger.info(f"✅ Groq initialized - Limit: {USER_DAILY_LIMIT} req/day")
            elif provider == 'openai':
                ai_client = OpenAIClient()
                logger.info(f"✅ OpenAI initialized - Limit: {USER_DAILY_LIMIT} req/day")
            else:
                raise ValueError(f"Unknown AI provider: {provider}")
        else:
            ai_client = OllamaClient()
            logger.info("✅ Ollama initialized (development)")

    except Exception as e:
        logger.error(f"❌ Error initializing AI: {e}")
        ai_client = None


async def assistant_reply(message: str, user_id: Optional[str] = None) -> dict:
    """Generate AI assistant reply with rate limiting"""
    try:
        if not ai_client:
            init_ai()

        if not ai_client:
            return {
                'success': False,
                'response': "AI assistant is not configured.",
                'remaining': 0
            }
```

```
    # Check rate limit (production only)
    if ENVIRONMENT == 'production' and user_id:
        is_allowed, remaining = rate_limiter.check_limit(user_id)

        if not is_allowed:
            return {
                'success': False,
                'response': f"Daily limit of {USER_DAILY_LIMIT} requests reached. Try tomorrow.",
                'remaining': 0
            }
    else:
        remaining = rate_limiter.get_remaining(user_id) if user_id else USER_DAILY_LIMIT

    # AI context
    context = """You are a helpful assistant.
Keep responses concise and friendly (under 500 tokens)."""

    # Get AI response
    response = await ai_client.chat(message, context)

    return {
        'success': True,
        'response': response,
        'remaining': remaining
    }

except Exception as e:
    logger.error(f"❌ Error in assistant_reply: {e}")
    return {
        'success': False,
        'response': "I'm having trouble right now. Please try again.",
        'remaining': 0
    }


# Initialize on import
init_ai()
```

Flask Route Examples

File: routes/chat.py

python

```python
from flask import Blueprint, request, jsonify
from services.ai_client import assistant_reply, rate_limiter, USER_DAILY_LIMIT
import logging

chat_bp = Blueprint('chat', __name__)
logger = logging.getLogger(__name__)

@chat_bp.route('/api/chat', methods=['POST'])
async def chat():
    """Chat endpoint with rate limiting"""
    try:
        data = request.get_json()

        if not data or 'message' not in data:
            return jsonify({
                'success': False,
                'error': 'Message is required'
            }), 400

        message = data['message'].strip()

        if not message:
            return jsonify({
                'success': False,
                'error': 'Message cannot be empty'
            }), 400

        # Get user identifier
        user_id = data.get('user_email') or data.get('session_id') or request.remote_addr

        # Get AI response
        result = await assistant_reply(message, user_id)

        if result['success']:
            return jsonify({
                'success': True,
                'response': result['response'],
                'remaining_requests': result['remaining']
            }), 200
        else:
            status_code = 429 if 'limit' in result['response'].lower() else 500
            return jsonify({
                'success': False,
```

```
            'error': result['response'],
            'remaining_requests': result['remaining']
        }), status_code

    except Exception as e:
        logger.error(f"Chat endpoint error: {e}")
        return jsonify({
            'success': False,
            'error': 'Internal server error'
        }), 500


@chat_bp.route('/api/chat/remaining', methods=['GET'])
def get_remaining():
    """Check remaining requests for user"""
    try:
        user_id = request.args.get('user_id') or request.remote_addr
        remaining = rate_limiter.get_remaining(user_id)

        return jsonify({
            'success': True,
            'remaining': remaining,
            'limit': USER_DAILY_LIMIT
        }), 200

    except Exception as e:
        logger.error(f"Error getting remaining: {e}")
        return jsonify({
            'success': False,
            'error': 'Internal server error'
        }), 500
```

File: app.py

python

```python
from flask import Flask, render_template
from flask_cors import CORS
from routes.chat import chat_bp
import logging

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

app = Flask(__name__)
CORS(app)

# Register blueprints
app.register_blueprint(chat_bp)

@app.route('/')
def index():
    return render_template('index.html')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

Environment Variables

Development (.env)

```bash
# Environment
ENVIRONMENT=development

# AI Configuration (Local)
AI_PROVIDER=ollama
OLLAMA_BASE_URL=http://localhost:11434
OLLAMA_MODEL=llama3.1:latest

# Rate Limiting
USER_DAILY_LIMIT=500

# Database (Local)
DATABASE_URL=sqlite:///local.db
```

Production (.env for Render/Railway)

```bash
# Environment
ENVIRONMENT=production

# AI Configuration (Cloud)
AI_PROVIDER=groq
GROQ_API_KEY=your-groq-api-key-here
# Or for OpenAI:
# AI_PROVIDER=openai
# OPENAI_API_KEY=your-openai-key-here

# Rate Limiting
USER_DAILY_LIMIT=500

# Database (Production)
DATABASE_URL=postgresql://user:pass@host:5432/dbname

# Flask
FLASK_ENV=production
SECRET_KEY=your-secret-key-here
```

.env.example (Template for team)

```bash
# Copy this to .env and fill in your values

# Environment (development or production)
ENVIRONMENT=development

# AI Provider (ollama, groq, or openai)
AI_PROVIDER=ollama

# Ollama (for local development)
OLLAMA_BASE_URL=http://localhost:11434
OLLAMA_MODEL=llama3.1:latest

# Groq API (for production - get free key at https://console.groq.com)
GROQ_API_KEY=

# OpenAI API (optional backup - get key at https://platform.openai.com)
OPENAI_API_KEY=

# Rate Limiting (requests per user per day)
USER_DAILY_LIMIT=500

# Database
DATABASE_URL=sqlite:///local.db

# Flask
FLASK_ENV=development
SECRET_KEY=change-this-in-production
```

Project Structure

```
your-project/
│
├── .vscode/                    # VS Code settings
│   └── ai-config.yaml          # VS Code AI assistant config
│
├── app.py                      # Main Flask application
├── requirements.txt            # Python dependencies
├── .env                        # Environment variables (don't commit!)
├── .env.example                # Template for .env
├── .gitignore                  # Git ignore file
├── README.md                   # Project documentation
│
├── services/                   # Business logic
│   ├── __init__.py
│   └── ai_client.py            # AI integration (Ollama/Groq/OpenAI)
│
├── routes/                     # Flask routes/blueprints
│   ├── __init__.py
│   ├── chat.py                 # Chat endpoints
│   └── api.py                  # Other API endpoints
│
├── models/                     # Database models
│   ├── __init__.py
│   └── database.py             # SQLAlchemy models
│
├── static/                     # Frontend static files
│   ├── css/
│   │   └── styles.css
│   ├── js/
│   │   └── app.js
│   └── images/
│
├── templates/                  # HTML templates
│   ├── index.html
│   ├── chat.html
│   └── base.html
│
└── tests/                      # Unit tests
    ├── __init__.py
    ├── test_ai_client.py
    └── test_routes.py
```

Cost Comparison Tables

AI API Costs

Provider	Model	Input (per 1M tokens)	Output (per 1M tokens)	Free Tier
Groq	llama-3.1-8b-instant	$0.05	$0.08	14.4K req/day FREE
OpenAI	gpt-4o-mini	$0.15	$0.60	$5 credit (expires)
OpenAI	gpt-4o	$2.50	$10.00	No free tier
Anthropic	Claude Haiku	$0.25	$1.25	No free tier
Local Ollama	Any model	$0	$0	Unlimited FREE

Monthly Cost Examples (Chat UI)

Scenario 1: Small Project (100 users/day)

- 100 users × 5 messages = 500 messages/day = 15,000/month
- Average 200 tokens per message = 3M tokens/month

Provider	Monthly Cost
Groq	$0 (within free tier)
OpenAI (gpt-4o-mini)	~$0.45
Local Ollama	$0

Scenario 2: Medium Project (1,000 users/day)

- 1,000 users × 5 messages = 5,000 messages/day = 150,000/month
- Average 200 tokens per message = 30M tokens/month

Provider	Monthly Cost
Groq	~$1.50 (slightly over free tier)
OpenAI (gpt-4o-mini)	~$4.50
Local Ollama	$0 (but needs powerful server)

Scenario 3: Large Project (5,000 users/day)

- 5,000 users × 5 messages = 25,000 messages/day = 750,000/month
- Average 200 tokens per message = 150M tokens/month

Provider	Monthly Cost
Groq	~$7.50
OpenAI (gpt-4o-mini)	~$22.50

Hosting Costs

Platform	Free Tier	After Free Tier
GitHub Pages	Unlimited	Always free
Netlify	100GB/mo	$19/mo (Pro)
Vercel	100GB/mo	$20/mo (Pro)
Render	750 hours/mo	$7/mo per service
Railway	$5 credit/mo	Pay-as-you-go
Fly.io	3 VMs free	$1.94/mo per VM

Database Costs

Provider	Free Tier	After Free Tier
Neon PostgreSQL	10GB	$19/mo (Launch)
Supabase	500MB	$25/mo (Pro)
MongoDB Atlas	512MB	$9/mo (M2)
PlanetScale	5GB	$29/mo (Scaler)
SQLite	Unlimited	Always free (local file)

Total Monthly Costs by Setup

Setup 1: Landing Page Only

- Hosting: GitHub Pages (free)
- Forms: Formspree (free)
- **Total: $0/month**

Setup 2: Chat UI (Small - 100 users/day)

- Frontend: Netlify (free)
- Backend: Render (free)
- Database: Neon PostgreSQL (free)
- AI: Groq (free)
- **Total: $0/month**

Setup 3: Chat UI (Medium - 1,000 users/day)

- Frontend: Netlify (free)
- Backend: Render (free or $7/mo)
- Database: Neon PostgreSQL (free)
- AI: Groq ($1.50/mo)
- **Total: $1.50-8.50/month**

Setup 4: Production App (5,000 users/day)

- Frontend: Vercel (free)
- Backend: Render ($7/mo)
- Database: Neon PostgreSQL (free or $19/mo if >10GB)
- AI: Groq ($7.50/mo)
- **Total: $14.50-33.50/month**

MCP Servers Reference

What Are MCP Servers?

MCP = Model Context Protocol

A standard created by Anthropic that allows AI assistants to connect to external tools and data sources. Think of them as plugins or extensions.

Popular MCP Servers

MCP Server	What It Does	Use Case
memory-mcp	Long-term memory across sessions	Remember project context
filesystem-mcp	Read/write files	Auto-update config files
github-mcp	GitHub API access	Create issues, PRs automatically
postgres-mcp	PostgreSQL queries	Query your database
brave-search-mcp	Web search	Research while coding
puppeteer-mcp	Browser automation	Test websites
slack-mcp	Slack integration	Send notifications
google-drive-mcp	Google Drive access	Read/write Drive files

How to Add MCP Servers

Example 1: Add Memory

```yaml
mcpServers:
  - uses: anthropic/memory-mcp
    config:
      max_memory_mb: 100
```

Example 2: Add GitHub Integration

```yaml
mcpServers:
  - uses: github-mcp
    config:
      token: ${GITHUB_TOKEN}
      repos:
        - username/repo-name
```

Example 3: Multiple MCP Servers

```yaml
mcpServers:
  - uses: anthropic/memory-mcp
  - uses: filesystem-mcp
    config:
      allowed_directories:
        - ~/projects
  - uses: github-mcp
    config:
      token: ${GITHUB_TOKEN}
```

Should You Use MCP Servers?

NO (Keep empty) if:

- You're just starting out
- Basic coding assistance is enough
- You don't need persistent memory
- You want to keep things simple

YES if:

- You want AI to remember past conversations
- You want AI to interact with tools automatically
- You're building complex workflows
- Basic functionality isn't enough

Recommendation: Start with `mcpServers: []` (empty), add them later when needed

---

Quick Reference Commands

Ollama Commands

```bash
# Install Ollama (Mac/Linux)
curl -fsSL https://ollama.com/install.sh | sh

# Install Ollama (Windows)
# Download from: https://ollama.com/download

# Pull models
ollama pull llama3.1:latest
ollama pull qwen2.5-coder:7b
ollama pull deepseek-coder-v2:16b
ollama pull codellama:13b

# List installed models
ollama list

# Run Ollama server
ollama serve

# Test a model
ollama run llama3.1

# Remove a model
ollama rm llama3.1
```

Python/Flask Commands

bash

```
# Create virtual environment
python -m venv venv

# Activate virtual environment (Mac/Linux)
source venv/bin/activate

# Activate virtual environment (Windows)
venv\Scripts\activate

# Install dependencies
pip install -r requirements.txt

# Create requirements.txt
pip freeze > requirements.txt

# Run Flask app
python app.py

# Run Flask app with auto-reload
flask run --reload

# Run tests
pytest
```

Git Commands

bash

```bash
# Initialize repository
git init

# Add all files
git add .

# Commit changes
git commit -m "Initial commit"

# Add remote repository
git remote add origin https://github.com/username/repo.git

# Push to GitHub
git push -u origin main

# Create new branch
git checkout -b feature-name

# Switch branches
git checkout main

# Pull latest changes
git pull origin main

# Check status
git status
```

Docker Commands (Optional)

```bash
# Build Docker image
docker build -t my-flask-app .

# Run Docker container
docker run -p 5000:5000 my-flask-app

# Run with environment variables
docker run -p 5000:5000 --env-file .env my-flask-app

# Stop container
docker stop container-id

# List running containers
docker ps

# List all containers
docker ps -a

# Remove container
docker rm container-id
```

Database Commands

```bash
# SQLite
sqlite3 local.db

# Create tables (using Flask-Migrate)
flask db init
flask db migrate -m "Initial migration"
flask db upgrade

# PostgreSQL (psql)
psql -U username -d database_name

# MongoDB (mongosh)
mongosh "mongodb://localhost:27017"
```

Python Requirements.txt

File: requirements.txt

```
# Flask Framework
Flask==3.0.0
Flask-CORS==4.0.0
Flask-Migrate==4.0.5
Flask-SQLAlchemy==3.1.1

# Database
SQLAlchemy==2.0.23
psycopg2-binary==2.9.9  # PostgreSQL
pymongo==4.6.1  # MongoDB (optional)

# AI Libraries
openai==1.6.1  # For OpenAI API
groq==0.4.1  # For Groq API
httpx==0.25.2  # For Ollama

# Utilities
python-dotenv==1.0.0
requests==2.31.0

# Development
pytest==7.4.3
black==23.12.1
flake8==6.1.0
```

Install with:

```bash
pip install -r requirements.txt
```

Example Frontend (Chat UI)

File: static/js/app.js

javascript

```javascript
// Chat UI JavaScript
class ChatApp {
    constructor() {
        this.chatForm = document.getElementById('chat-form');
        this.chatInput = document.getElementById('chat-input');
        this.chatMessages = document.getElementById('chat-messages');
        this.remainingEl = document.getElementById('remaining-requests');

        this.init();
    }

    init() {
        this.chatForm.addEventListener('submit', (e) => this.handleSubmit(e));
        this.updateRemaining();
    }

    async handleSubmit(e) {
        e.preventDefault();

        const message = this.chatInput.value.trim();
        if (!message) return;

        // Add user message to UI
        this.addMessage(message, 'user');
        this.chatInput.value = '';

        // Show loading
        const loadingId = this.addMessage('Thinking...', 'assistant', true);

        try {
            const response = await fetch('/api/chat', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({
                    message: message,
                    session_id: this.getSessionId()
                })
            });

            const data = await response.json();
```

```
        // Remove loading message
        document.getElementById(loadingId).remove();

        if (data.success) {
            this.addMessage(data.response, 'assistant');
            this.updateRemaining(data.remaining_requests);
        } else {
            this.addMessage(`Error: ${data.error}`, 'error');
        }

    } catch (error) {
        document.getElementById(loadingId).remove();
        this.addMessage('Failed to get response. Please try again.', 'error');
        console.error('Chat error:', error);
    }
}

addMessage(text, sender, isLoading = false) {
    const messageId = `msg-${Date.now()}`;
    const messageEl = document.createElement('div');
    messageEl.id = messageId;
    messageEl.className = `message message-${sender}`;
    messageEl.textContent = text;

    if (isLoading) {
        messageEl.classList.add('loading');
    }

    this.chatMessages.appendChild(messageEl);
    this.chatMessages.scrollTop = this.chatMessages.scrollHeight;

    return messageId;
}

async updateRemaining(count = null) {
    if (count !== null) {
        this.remainingEl.textContent = `Remaining: ${count}`;
        return;
    }

    try {
        const response = await fetch(`/api/chat/remaining?user_id=${this.getSessionId()}`);
        const data = await response.json();

        if (data.success) {
```

```
            this.remainingEl.textContent = `Remaining: ${data.remaining}/${data.limit}`;
        }
    } catch (error) {
        console.error('Failed to get remaining requests:', error);
    }
}

getSessionId() {
    let sessionId = localStorage.getItem('session_id');
    if (!sessionId) {
        sessionId = `session-${Date.now()}-${Math.random()}`;
        localStorage.setItem('session_id', sessionId);
    }
    return sessionId;
}
}

// Initialize app when DOM is ready
document.addEventListener('DOMContentLoaded', () => {
    new ChatApp();
});
```

File: templates/index.html

html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>AI Chat Assistant</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/styles.css') }}">
</head>
<body>
    <div class="container">
        <header>
            <h1>AI Chat Assistant</h1>
            <p id="remaining-requests">Loading...</p>
        </header>

        <div id="chat-messages" class="chat-messages">
            <!-- Messages appear here -->
        </div>

        <form id="chat-form" class="chat-form">
            <input
                type="text"
                id="chat-input"
                placeholder="Type your message..."
                autocomplete="off"
                required
            >
            <button type="submit">Send</button>
        </form>
    </div>

    <script src="{{ url_for('static', filename='js/app.js') }}"></script>
</body>
</html>
```

File: static/css/styles.css

css

```css
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
    background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
    min-height: 100vh;
    display: flex;
    justify-content: center;
    align-items: center;
    padding: 20px;
}

.container {
    width: 100%;
    max-width: 800px;
    height: 600px;
    background: white;
    border-radius: 20px;
    box-shadow: 0 20px 60px rgba(0, 0, 0, 0.3);
    display: flex;
    flex-direction: column;
}

header {
    padding: 20px;
    border-bottom: 2px solid #f0f0f0;
    display: flex;
    justify-content: space-between;
    align-items: center;
}

h1 {
    font-size: 1.5em;
    color: #333;
}

#remaining-requests {
    color: #666;
    font-size: 0.9em;
```

```
}

.chat-messages {
    flex: 1;
    padding: 20px;
    overflow-y: auto;
    display: flex;
    flex-direction: column;
    gap: 15px;
}

.message {
    max-width: 70%;
    padding: 12px 16px;
    border-radius: 12px;
    word-wrap: break-word;
}

.message-user {
    align-self: flex-end;
    background: #667eea;
    color: white;
}

.message-assistant {
    align-self: flex-start;
    background: #f0f0f0;
    color: #333;
}

.message-error {
    align-self: center;
    background: #ff6b6b;
    color: white;
}

.message.loading {
    opacity: 0.6;
    animation: pulse 1.5s infinite;
}

@keyframes pulse {
    0%, 100% { opacity: 0.6; }
    50% { opacity: 1; }
}
```

```
.chat-form {
    padding: 20px;
    border-top: 2px solid #f0f0f0;
    display: flex;
    gap: 10px;
}

#chat-input {
    flex: 1;
    padding: 12px 16px;
    border: 2px solid #e0e0e0;
    border-radius: 12px;
    font-size: 1em;
    outline: none;
    transition: border-color 0.3s;
}

#chat-input:focus {
    border-color: #667eea;
}

button {
    padding: 12px 30px;
    background: #667eea;
    color: white;
    border: none;
    border-radius: 12px;
    font-size: 1em;
    font-weight: 600;
    cursor: pointer;
    transition: background 0.3s;
}

button:hover {
    background: #5568d3;
}

button:active {
    transform: scale(0.98);
}

/* Mobile responsive */
@media (max-width: 768px) {
    .container {
        height: 100vh;
```

```
    height: 100vh;
    border-radius: 0;
  }

  .message {
    max-width: 85%;
  }
}
```

.gitignore File

File: .gitignore

```
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
venv/
env/
ENV/
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
*.egg-info/
.installed.cfg
*.egg

# Environment variables
.env
.env.local
.env.*.local

# IDE
.vscode/
.idea/
*.swp
*.swo
*~

# Database
*.db
*.sqlite
*.sqlite3
local.db

# Logs
*.log
```

```
*.log
logs/

# OS
.DS_Store
Thumbs.db

# Flask
instance/
.webassets-cache

# Testing
.pytest_cache/
.coverage
htmlcov/

# Docker
docker-compose.override.yml

# Node (if using)
node_modules/
npm-debug.log
yarn-error.log
```

Deployment Guide

Deploy to Render (FREE)

Step 1: Prepare Your Code

```bash
# Make sure you have these files:
# - app.py
# - requirements.txt
# - .env.example (don't include .env!)
```

Step 2: Create Render Account

- Go to https://render.com
- Sign up with GitHub

Step 3: Create New Web Service

- Click "New +" → "Web Service"
- Connect your GitHub repository
- Configure:
  - **Name:** your-app-name
  - **Environment:** Python 3
  - **Build Command:** `pip install -r requirements.txt`
  - **Start Command:** `gunicorn app:app`
  - **Plan:** Free

Step 4: Add Environment Variables In Render dashboard, add:

```
ENVIRONMENT=production
AI_PROVIDER=groq
GROQ_API_KEY=your-key-here
USER_DAILY_LIMIT=500
DATABASE_URL=your-neon-url-here
SECRET_KEY=generate-random-string
```

Step 5: Deploy

- Click "Create Web Service"
- Wait for deployment (5-10 minutes)
- Your app will be live at `https://your-app-name.onrender.com`

Deploy to Railway ($5 credit/month)

Step 1: Create Railway Account

- Go to https://railway.app
- Sign up with GitHub

Step 2: Create New Project

- Click "New Project"
- Select "Deploy from GitHub repo"
- Choose your repository

Step 3: Add Environment Variables

- Go to "Variables" tab
- Add all environment variables

Step 4: Deploy

- Railway auto-deploys on push to main branch
- Get your URL from dashboard

Deploy Frontend to Netlify (FREE)

For Static Frontend Only:

Step 1: Create Netlify Account

- Go to https://netlify.com
- Sign up with GitHub

Step 2: Deploy

- Drag and drop your `static/` folder
- Or connect GitHub repository

Step 3: Configure

- Set build command if needed
- Add environment variables for API URLs

Troubleshooting

Common Issues

Issue: "Cannot connect to Ollama"

```bash
# Solution: Start Ollama server
ollama serve

# Or check if it's running
ps aux | grep ollama
```

Issue: "GROQ_API_KEY not set"

```bash
# Solution: Add to .env file
echo "GROQ_API_KEY=your-key-here" >> .env

# Or export in terminal
export GROQ_API_KEY=your-key-here
```

Issue: "Rate limit exceeded"

```python
# Solution: Wait 1 minute or increase USER_DAILY_LIMIT
USER_DAILY_LIMIT=1000  # in .env
```

Issue: "Database connection failed"

```bash
# Solution: Check DATABASE_URL format
# PostgreSQL: postgresql://user:pass@host:5432/dbname
# SQLite: sqlite:///local.db
```

Issue: "Module not found"

```bash
# Solution: Install missing package
pip install package-name

# Or reinstall all
pip install -r requirements.txt
```

Issue: "Port already in use"

```bash
# Solution: Kill process on port 5000
# Mac/Linux:
lsof -ti:5000 | xargs kill -9

# Windows:
netstat -ano | findstr :5000
taskkill /PID <PID> /F
```

Security Best Practices

API Keys

- ✅**DO:** Store in environment variables
- ✅**DO:** Use `.env` file (add to `.gitignore`)
- ❌**DON'T:** Commit to Git
- ❌**DON'T:** Hardcode in source files
- ❌**DON'T:** Expose in frontend code

Rate Limiting

- ✅**DO:** Implement per-user limits
- ✅**DO:** Set reasonable daily limits (400-500)
- ✅**DO:** Log violations
- ❌**DON'T:** Allow unlimited requests

Input Validation

- ✅**DO:** Validate all user inputs
- ✅**DO:** Sanitize before processing
- ✅**DO:** Set max message length
- ❌**DON'T:** Trust user input blindly

HTTPS

- ✅**DO:** Use HTTPS in production
- ✅**DO:** Let platforms handle SSL (Render/Netlify)
- ❌**DON'T:** Send sensitive data over HTTP

Database

- ✅**DO:** Use parameterized queries
- ✅**DO:** Hash passwords (bcrypt)
- ✅**DO:** Regular backups
- ❌**DON'T:** Store plain-text passwords

Performance Optimization

Caching

python

```python
from flask_caching import Cache

cache = Cache(app, config={'CACHE_TYPE': 'simple'})

@cache.cached(timeout=300)  # Cache for 5 minutes
def get_popular_responses():
    # Expensive operation
    pass
```

Database Indexing

python

```python
# Add indexes to frequently queried fields
class User(db.Model):
    email = db.Column(db.String(120), index=True, unique=True)
```

Async Operations

python

```python
# Use async for I/O-bound operations
async def fetch_multiple_apis():
    tasks = [
        fetch_api_1(),
        fetch_api_2(),
        fetch_api_3()
    ]
    results = await asyncio.gather(*tasks)
    return results
```

Testing

Unit Tests Example

File: `tests/test_ai_client.py`

python

```python
import pytest
from services.ai_client import RateLimiter, assistant_reply

def test_rate_limiter():
    limiter = RateLimiter(limit_per_day=2)

    # First request - should succeed
    allowed, remaining = limiter.check_limit("user1")
    assert allowed == True
    assert remaining == 1

    # Second request - should succeed
    allowed, remaining = limiter.check_limit("user1")
    assert allowed == True
    assert remaining == 0

    # Third request - should fail
    allowed, remaining = limiter.check_limit("user1")
    assert allowed == False
    assert remaining == 0

@pytest.mark.asyncio
async def test_assistant_reply():
    response = await assistant_reply("Hello", user_id="test_user")

    assert response['success'] == True
    assert 'response' in response
    assert 'remaining' in response
```

Run tests:

bash

```bash
pytest
pytest -v  # Verbose
pytest tests/test_ai_client.py  # Specific file
```

Resources & Links

Documentation

- **Flask:** https://flask.palletsprojects.com/
- **SQLAlchemy:** https://www.sqlalchemy.org/
- **Ollama:** https://ollama.com/
- **Groq:** https://console.groq.com/docs
- **OpenAI:** https://platform.openai.com/docs

Free Hosting

- **Render:** https://render.com
- **Railway:** https://railway.app
- **Netlify:** https://netlify.com
- **Vercel:** https://vercel.com
- **GitHub Pages:** https://pages.github.com

Free Databases

- **Neon PostgreSQL:** https://neon.tech
- **Supabase:** https://supabase.com
- **MongoDB Atlas:** https://www.mongodb.com/atlas
- **PlanetScale:** https://planetscale.com

API Keys (Get Free)

- **Groq:** https://console.groq.com
- **OpenAI:** https://platform.openai.com
- **Together AI:** https://api.together.xyz

Learning Resources

- **Flask Tutorial:** https://flask.palletsprojects.com/tutorial/
- **Python Docs:** https://docs.python.org/3/
- **MDN Web Docs:** https://developer.mozilla.org/

Next Steps

For Beginners

1. Set up Ollama locally
2. Create simple Flask app
3. Build basic chat UI
4. Test locally
5. Deploy to Render (free)

For Intermediate

1. Add user authentication
2. Implement chat history
3. Add file upload
4. Deploy with CI/CD
5. Monitor usage

For Advanced

1. Add Redis caching
2. Implement WebSockets
3. Multi-model support
4. Advanced analytics
5. Scale with Kubernetes

Summary Checklist

Development Setup

- [ ] Install Python 3.9+
- [ ] Install Ollama
- [ ] Pull Ollama models
- [ ] Create virtual environment
- [ ] Install dependencies
- [ ] Create .env file
- [ ] Test locally

Production Deployment

- [ ] Get Groq API key (free)
- [ ] Create Neon PostgreSQL database (free)
- [ ] Sign up for Render (free)
- [ ] Push code to GitHub
- [ ] Deploy to Render
- [ ] Add environment variables
- [ ] Test production deployment

Post-Deployment

- [ ] Monitor usage
- [ ] Check error logs
- [ ] Test rate limiting
- [ ] Backup database
- [ ] Update documentation

This guide contains everything discussed. Save it for offline reference!

Questions? Review the appropriate section above or refer to the official documentation links.