Architecture Document

AI Social Media Content Generator

Overview

This document outlines the architectural decisions, trade-offs, and scaling considerations for the AI Social Media Content Generator application. The application is built as a Next.js full-stack application that integrates with OpenAI's API to generate social media content and stores the results in PostgreSQL.

Architecture Decisions

1. Framework: Next.js with App Router

Decision: Use Next.js 14 with the App Router pattern.

Rationale:

- **Full-Stack Capabilities**: Next.js provides both frontend and backend in a single framework, reducing complexity
- **API Routes**: Built-in API route handlers eliminate the need for a separate backend server
- **Server Components**: Leverages React Server Components for better performance
- **TypeScript Support**: Excellent TypeScript integration out of the box
- **Deployment**: Easy deployment on Vercel or other platforms

Trade-offs:

- ✅ Pros: Single codebase, fast development, good performance
- ❌ Cons: Less flexibility than separate frontend/backend, vendor lock-in to Next.js patterns

---

2. Database: PostgreSQL with Prisma ORM

Decision: Use PostgreSQL as the database with Prisma as the ORM.

Rationale:

- **Relational Data**: PostgreSQL is ideal for structured data with relationships

- **ACID Compliance**: Ensures data integrity for critical operations
- **Prisma Benefits**: Type-safe queries, migrations, and excellent developer experience
- **Scalability**: PostgreSQL can handle significant scale with proper indexing
- **Hosted Options**: Easy to use with Supabase, Neon, or other managed services

Trade-offs:

- ✅ Pros: Strong consistency, type safety, mature ecosystem
- ❌ Cons: Can be overkill for simple use cases, requires connection pooling at scale

Schema Design:

```
model GeneratedPost {
  id          String   @id @default(cuid())
  prompt      String
  content     String   @db.Text
  platform    String?
  userId      String?
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt

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

Design Decisions:

- Use CUID for IDs (better than auto-increment for distributed systems)
- Store content as Text type (supports long-form content)
- Optional userId field for future multi-user support
- Indexes on userId and createdAt for efficient queries

---

3. AI Integration: OpenAI GPT-4o-mini

Decision: Use OpenAI's GPT-4o-mini model via their official SDK.

Rationale:

- **Cost-Effective**: GPT-4o-mini provides good quality at lower cost than GPT-4
- **Reliability**: OpenAI has robust infrastructure and API
- **Flexibility**: Easy to switch models or adjust parameters
- **Official SDK**: Well-maintained TypeScript SDK with proper error handling

Trade-offs:

- ✅ Pros: High-quality output, reliable API, good documentation
- ❌ Cons: External dependency, costs per request, potential rate limits

Implementation:

- System prompt for consistent output format
- Configurable maxTokens and temperature
- Platform-specific prompts for tailored content
- Proper error handling for rate limits and API failures

---

4. Error Handling Strategy

Decision: Implement comprehensive error handling at multiple layers.

Layers:

1. **Validation Layer**: Zod schema validation for API inputs
2. **API Layer**: Try-catch blocks with specific error types
3. **Client Layer**: User-friendly error messages

Error Types Handled:

- Validation errors (400)
- Rate limit errors (429)
- API errors (500)

- Database errors (500)

Trade-offs:

- ✅ Pros: Better user experience, easier debugging
- ❌ Cons: More code, potential information leakage (mitigated by generic messages)

---

5. State Management: React Hooks

Decision: Use React hooks (useState, useEffect) for local state management.

Rationale:

- **Simplicity**: No need for external state management for this scope
- **Built-in**: React hooks are sufficient for component-level state
- **Performance**: Minimal overhead, good for small to medium applications

Trade-offs:

- ✅ Pros: Simple, no additional dependencies
- ❌ Cons: Could become complex with more features (would need Context API or Zustand)

---

6. Styling: Tailwind CSS

Decision: Use Tailwind CSS for styling.

Rationale:

- **Rapid Development**: Utility-first approach speeds up UI development
- **Consistency**: Design system built-in
- **Bundle Size**: Only includes used styles
- **Modern**: Widely adopted, good documentation

Trade-offs:

- ✅ Pros: Fast development, consistent design, small bundle
- ❌ Cons: Can be verbose, requires learning utility classes

Scalability Considerations

Current Architecture Limitations

1. **Single Database Instance**: No read replicas or sharding
2. **No Caching**: Every request hits the database
3. **Synchronous API Calls**: OpenAI API calls block the request
4. **No Queue System**: No background job processing
5. **Single Server**: No horizontal scaling

Scaling Strategies

1. Database Scaling

Short-term (0-10K users):

- Use connection pooling (PgBouncer or Prisma's built-in pooling)
- Optimize queries with proper indexes
- Use read replicas for GET requests

Medium-term (10K-100K users):

- Implement database sharding by userId
- Use caching layer (Redis) for frequently accessed posts
- Consider time-series database for analytics

Long-term (100K+ users):

- Move to distributed database (CockroachDB, Vitess)
- Implement eventual consistency where appropriate
- Archive old posts to cold storage

2. API Scaling

Short-term:

- Add request rate limiting per user/IP
- Implement request queuing for OpenAI API calls

- Use edge functions for static content

Medium-term:

- Move OpenAI calls to background jobs (BullMQ, AWS SQS)
- Implement WebSocket for real-time updates
- Add CDN for static assets

Long-term:

- Microservices architecture:
    - Content Generation Service
    - User Service
    - Analytics Service
- API Gateway for routing and rate limiting
- Load balancing across multiple instances

3. Caching Strategy

Implementation:

- **Redis** for:
    - Frequently accessed posts (cache for 5-10 minutes)
    - Rate limit counters
    - Session data (if adding authentication)

Cache Invalidation:

- Invalidate on post creation/update
- TTL-based expiration
- Manual invalidation for critical updates

4. Background Job Processing

For OpenAI API Calls:

- Queue system (BullMQ, AWS SQS, or similar)
- Worker processes to handle generation

- WebSocket or polling for status updates
- Retry logic with exponential backoff

Benefits:

- Non-blocking API responses
- Better error handling
- Rate limit management
- Cost optimization (batch processing)

5. Monitoring and Observability

Essential Metrics:

- API response times
- OpenAI API latency and errors
- Database query performance
- Error rates by type
- User activity metrics

Tools:

- Application: Sentry for error tracking
- Performance: Datadog, New Relic, or Vercel Analytics
- Logs: Structured logging with Winston or Pino
- Database: pg_stat_statements for query analysis

Security Considerations

Current Implementation

1. **Input Validation**: Zod schemas validate all inputs
2. **SQL Injection**: Prisma prevents SQL injection
3. **API Key Security**: Environment variables for sensitive data
4. **Error Messages**: Generic error messages to prevent information leakage

Future Enhancements

1. **Authentication**: Add user authentication (NextAuth.js, Clerk)
2. **Authorization**: Role-based access control
3. **Rate Limiting**: Per-user rate limiting
4. **API Key Rotation**: Support for rotating OpenAI API keys
5. **Audit Logging**: Log all API calls and data access
6. **Content Moderation**: Filter inappropriate content before storage

---

Cost Optimization

Current Costs

- **OpenAI API**: ~$0.15 per 1M input tokens, ~$0.60 per 1M output tokens (GPT-4o-mini)
- **Database**: Varies by provider (Supabase free tier, Neon free tier available)
- **Hosting**: Vercel free tier for small projects

Optimization Strategies

1. **Caching**: Reduce redundant API calls
2. **Token Optimization**: Tune maxTokens based on platform
3. **Batch Processing**: Group similar requests
4. **Model Selection**: Use cheaper models for simple tasks
5. **Database**: Archive old posts, use cheaper storage tiers

---

Future Enhancements

Phase 1 (Short-term)

- User authentication and authorization
- Post editing and regeneration
- Export posts (copy, download)
- Basic analytics (views, generations per day)

Phase 2 (Medium-term)

- Multi-language support
- Content templates
- Scheduled posting integration
- Advanced filtering and search

Phase 3 (Long-term)

- Multi-AI provider support (Anthropic, etc.)
- A/B testing for content
- Team collaboration features
- API for third-party integrations

Conclusion

The current architecture is designed for rapid development and initial scalability. It prioritizes:

1. **Developer Experience**: Easy to set up, maintain, and extend
2. **Type Safety**: TypeScript and Prisma ensure fewer runtime errors
3. **Performance**: Next.js optimizations and efficient database queries
4. **Reliability**: Comprehensive error handling and validation

As the application scales, the suggested enhancements (caching, queuing, microservices) can be incrementally adopted based on actual usage patterns and requirements.

Document Version: 1.0
Last Updated: November 2024