Complete Development Plan for Personal Note Manager MCP Server

This comprehensive development plan guides you through building a Personal Note Manager using the Model Context Protocol (MCP), designed for developers learning MCP server development. The project progresses from basic setup to advanced features, incorporating current best practices and real-world implementation patterns from 2024-2025.

Project overview and learning objectives

The Personal Note Manager MCP server teaches core MCP concepts through practical implementation. You'll build a fully functional note-taking system that integrates with Claude Desktop and other MCP-compatible clients, Anthropic learning essential patterns for production-ready MCP development. Zep +5 **The complete project requires approximately 40-50 hours** of focused development time, suitable for completion over 2-3 weeks.

Section 1: Environment setup and MCP fundamentals

Difficulty: Easy
Estimated Time: 3-4 hours
Purpose: Establish development environment and understand MCP architecture

Technical requirements

Install the following dependencies for your development environment: Model Context Protocol

```
bash

# Node.js/TypeScript requirements
Node.js 18+ LTS
npm 9+
TypeScript 5.3+

# Python alternative (if preferred)
Python 3.10+
uv package manager (pip install uv)

# Development tools
MCP Inspector (npm install -g @modelcontextprotocol/inspector)
VS Code with TypeScript/Python extensions
Git for version control
```

github

Step-by-step implementation

Step 1: Initialize project structure (30 minutes)

Create the following directory structure:

```
notes-mcp-server/
├── src/
│   ├── index.ts
│   ├── server.ts
│   ├── storage/
│   ├── tools/
│   ├── types/
│   └── utils/
├── data/
├── tests/
├── package.json
├── tsconfig.json
└── README.md
```

Step 2: Configure TypeScript and dependencies (45 minutes)

Create `package.json`:

```
json
```

```
{
  "name": "notes-mcp-server",
  "version": "1.0.0",
  "type": "module",
  "main": "./build/index.js",
  "scripts": {
    "build": "tsc",
    "dev": "tsx watch src/index.ts",
    "start": "node build/index.js",
    "inspector": "npx @modelcontextprotocol/inspector build/index.js"
  },
  "dependencies": {
    "@modelcontextprotocol/sdk": "^1.17.4",
    "zod": "^3.22.0",
    "uuid": "^9.0.0"
  },
  "devDependencies": {
    "@types/node": "^20.0.0",
    "typescript": "^5.3.0",
    "tsx": "^4.0.0"
  }
}
```

Configure `tsconfig.json`:

json

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "NodeNext",
    "moduleResolution": "NodeNext",
    "outDir": "./build",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "build"]
}
```

Step 3: Create basic MCP server (1 hour)

Implement src/index.ts: Medium Fka

```typescript
#!/usr/bin/env node
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/stdio.js";

async function main() {
  const server = new McpServer({
    name: "notes-server",
    version: "1.0.0"
  });

  // Server will be configured in next sections

  const transport = new StdioServerTransport();
  await server.connect(transport);

  // Critical: Use console.error for logging in STDIO mode
  console.error("Notes MCP server started");
}

main().catch(console.error);
```

github

Step 4: Test basic setup (30 minutes)

Run the MCP Inspector to verify server initialization: Model Context Protocol +2

```bash
npm run build
npm run inspector
```

github

Common pitfalls to avoid

1. **Never write to stdout in STDIO mode** - This corrupts JSON-RPC messages Model Context Protocol Stainless

2. **Use absolute paths in configuration** - Relative paths cause connection failures Model Context Protocol Arsturn
3. **Check Node.js version compatibility** - MCP SDK requires Node.js 18+ Arsturn
4. **Validate JSON syntax in config files** - Malformed JSON is the #1 setup issue Arsturn

Section 2: Note data model and storage layer

Difficulty: Easy
Estimated Time: 4-5 hours
Purpose: Implement persistent storage with proper error handling

Implementation details

Step 1: Define Note type system (45 minutes)

Create `src/types/note.ts`:

```typescript
import { z } from "zod";

export const NoteSchema = z.object({
  id: z.string().uuid(),
  title: z.string().min(1).max(200),
  content: z.string(),
  tags: z.array(z.string()).default([]),
  created: z.string().datetime(),
  updated: z.string().datetime(),
  metadata: z.record(z.any()).optional()
});

export type Note = z.infer<typeof NoteSchema>;

export const CreateNoteSchema = z.object({
  title: z.string().min(1).max(200),
  content: z.string(),
  tags: z.array(z.string()).optional()
});

export type CreateNoteInput = z.infer<typeof CreateNoteSchema>;
```

github

Step 2: Implement JSON file storage (2 hours)

Create `src/storage/jsonStorage.ts`:

```typescript
```

```typescript
import { promises as fs } from 'fs';
import path from 'path';
import { Note, NoteSchema } from '../types/note.js';
import { v4 as uuidv4 } from 'uuid';

export class JsonNotesStorage {
  private dataPath: string;
  private notes: Map<string, Note> = new Map();
  private initialized = false;

  constructor(dataPath: string = './data/notes.json') {
    this.dataPath = path.resolve(dataPath);
  }

  async initialize(): Promise<void> {
    if (this.initialized) return;

    const dir = path.dirname(this.dataPath);
    try {
      await fs.access(dir);
    } catch {
      await fs.mkdir(dir, { recursive: true });
    }

    await this.loadNotes();
    this.initialized = true;
  }

  private async loadNotes(): Promise<void> {
    try {
      const data = await fs.readFile(this.dataPath, 'utf8');
      const parsed = JSON.parse(data);

      if (Array.isArray(parsed)) {
        for (const note of parsed) {
          const validated = NoteSchema.parse(note);
          this.notes.set(validated.id, validated);
        }
      }
    } catch (error: any) {
      if (error.code !== 'ENOENT') {
        console.error('Error loading notes:', error);
      }
```

```
  }
}

private async saveNotes(): Promise<void> {
  const notesArray = Array.from(this.notes.values());
  const tempPath = `${this.dataPath}.tmp`;

  await fs.writeFile(
    tempPath,
    JSON.stringify(notesArray, null, 2),
    'utf8'
  );
  await fs.rename(tempPath, this.dataPath);
}

async createNote(title: string, content: string, tags?: string[]): Promise<Note> {
  const now = new Date().toISOString();
  const note: Note = {
    id: uuidv4(),
    title,
    content,
    tags: tags || [],
    created: now,
    updated: now
  };

  this.notes.set(note.id, note);
  await this.saveNotes();
  return note;
}

async getNote(id: string): Promise<Note | null> {
  return this.notes.get(id) || null;
}

async getAllNotes(): Promise<Note[]> {
  return Array.from(this.notes.values());
}

async deleteNote(id: string): Promise<boolean> {
  const existed = this.notes.delete(id);
  if (existed) {
    await this.saveNotes();
  }
```

```
    return existed;
  }
}
```

Step 3: Add error handling (1 hour)

Create `src/utils/errors.ts`:

```typescript
export class NotesError extends Error {
  constructor(
    public code: string,
    message: string,
    public details?: any
  ) {
    super(message);
    this.name = 'NotesError';
  }
}

export function handleStorageError(error: any): string {
  console.error('Storage error:', error);

  if (error.code === 'EACCES') {
    return 'Permission denied accessing notes file';
  }
  if (error.code === 'ENOSPC') {
    return 'No space left on device';
  }
  if (error instanceof SyntaxError) {
    return 'Corrupted notes database';
  }

  return 'An unexpected storage error occurred';
}
```

Testing approaches

Create `tests/storage.test.ts` to verify storage operations: FastMCP

```typescript
```

```typescript
import { JsonNotesStorage } from '../src/storage/jsonStorage';
import { beforeEach, describe, expect, it } from '@jest/globals';

describe('JsonNotesStorage', () => {
  let storage: JsonNotesStorage;

  beforeEach(async () => {
    storage = new JsonNotesStorage('./test-data/notes.json');
    await storage.initialize();
  });

  it('should create and retrieve notes', async () => {
    const note = await storage.createNote('Test', 'Content');
    expect(note.title).toBe('Test');

    const retrieved = await storage.getNote(note.id);
    expect(retrieved).toEqual(note);
  });
});
```

Section 3: Implementing CRUD tools

Difficulty: Medium

Estimated Time: 6-8 hours

Purpose: Create MCP tools for note operations with proper validation

Tool implementation patterns

Step 1: Create tool registration system (2 hours)

Update `src/server.ts`:

```typescript
```

```
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { z } from "zod";
import { JsonNotesStorage } from "./storage/jsonStorage.js";
import { NotesError, handleStorageError } from "./utils/errors.js";

export class NotesServer {
  private server: McpServer;
  private storage: JsonNotesStorage;

  constructor() {
    this.server = new McpServer({
      name: "notes-server",
      version: "1.0.0"
    });

    this.storage = new JsonNotesStorage();
  }

  async initialize(): Promise<void> {
    await this.storage.initialize();
    this.registerTools();
  }

  private registerTools(): void {
    // Create Note Tool
    this.server.registerTool(
      "create_note",
      {
        title: "Create Note",
        description: "Create a new note with title and content",
        inputSchema: {
          title: z.string().min(1).max(200).describe("Note title"),
          content: z.string().describe("Note content"),
          tags: z.array(z.string()).optional().describe("Optional tags")
        }
      },
      async ({ title, content, tags }) => {
        try {
          const note = await this.storage.createNote(title, content, tags);
          return {
            content: [{
              type: "text",
              text: JSON.stringify({
```

```
          message: `Created note "${title}"`,
          noteId: note.id
        }, null, 2)
      }]
    };
  } catch (error) {
    return {
      content: [{
        type: "text",
        text: handleStorageError(error)
      }],
      isError: true
    };
  }
 }
);

// List Notes Tool
this.server.registerTool(
 "list_notes",
 {
  title: "List Notes",
  description: "List all notes with optional tag filtering",
  inputSchema: {
   tag: z.string().optional().describe("Filter by tag"),
   limit: z.number().min(1).max(100).default(20).describe("Max results")
  }
 },
 async ({ tag, limit }) => {
  try {
   let notes = await this.storage.getAllNotes();

   if (tag) {
    notes = notes.filter(n => n.tags.includes(tag));
   }

   notes = notes.slice(0, limit);

   const summary = notes.map(n => ({
    id: n.id,
    title: n.title,
    tags: n.tags,
    created: n.created
   }));
```

```
        return {
          content: [{
            type: "text",
            text: JSON.stringify(summary, null, 2)
          }]
        };
      } catch (error) {
        return {
          content: [{
            type: "text",
            text: handleStorageError(error)
          }],
          isError: true
        };
      }
    }
  );

  // Continue with get_note, update_note, delete_note...
}

getServer(): McpServer {
  return this.server;
}
}
```

github

Step 2: Implement remaining CRUD tools (3 hours)

Add the remaining tools following the same pattern:

- `get_note`: Retrieve specific note by ID
- `update_note`: Modify existing note
- `delete_note`: Remove note from storage
- `search_notes`: Full-text search capability

Step 3: Add input validation (1 hour)

Create `src/utils/validation.ts`:

```
typescript
```

```
import { z } from "zod";

export function validateAndSanitize<T>(
  input: unknown,
  schema: z.ZodSchema<T>
): T {
  try {
    return schema.parse(input);
  } catch (error) {
    if (error instanceof z.ZodError) {
      const issues = error.issues.map(i =>
        `${i.path.join('.')}: ${i.message}`
      ).join(', ');
      throw new NotesError('VALIDATION_ERROR', `Invalid input: ${issues}`);
    }
    throw error;
  }
}

export function sanitizeHtml(text: string): string {
  return text
    .replace(/</g, '&lt;')
    .replace(/>/g, '&gt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#x27;')
    .replace(/\//g, '&#x2F;');
}
```

Testing with MCP Inspector

Test each tool thoroughly using the MCP Inspector: Model Context Protocol +3

```
bash

npm run build
npm run inspector
```

github

Test scenarios:

1. Create notes with various title/content lengths
2. Test tag filtering in list_notes

3. Verify error handling with invalid IDs
4. Test concurrent operations
5. Validate input sanitization

Section 4: Advanced features - resources and prompts

Difficulty: Medium

Estimated Time: 5-6 hours

Purpose: Implement MCP resources and prompts for enhanced functionality Medium github

Implementing resources

Step 1: Create resource handlers (2 hours)

Add to src/server.ts: Medium

```typescript
```

```
private registerResources(): void {
  // Register note resource pattern
  this.server.registerResource(
    "note:///{id}",
    {
      title: "Note Content",
      description: "Access note content by ID"
    },
    async ({ id }) => {
      const note = await this.storage.getNote(id);
      if (!note) {
        throw new NotesError('NOT_FOUND', `Note ${id} not found`);
      }

      return {
        uri: `note:///${id}`,
        mimeType: "text/markdown",
        content: `# ${note.title}\n\n${note.content}\n\nTags: ${note.tags.join(', ')}`
      };
    }
  );

  // Register notes list resource
  this.server.registerResource(
    "notes:///list",
    {
      title: "All Notes",
      description: "List of all available notes"
    },
    async () => {
      const notes = await this.storage.getAllNotes();
      return {
        uri: "notes:///list",
        mimeType: "application/json",
        content: JSON.stringify(notes, null, 2)
      };
    }
  );
}
```

github

Step 2: Implement prompt templates (2 hours)

```typescript
private registerPrompts(): void {
  this.server.registerPrompt(
    "summarize_notes",
    {
      title: "Summarize Notes",
      description: "Generate a summary of selected notes"
    },
    async ({ tag }) => {
      const notes = await this.storage.getAllNotes();
      const filtered = tag ?
        notes.filter(n => n.tags.includes(tag)) : notes;

      const noteContent = filtered
        .map(n => `Title: ${n.title}\n${n.content}`)
        .join('\n---\n');

      return {
        messages: [
          {
            role: "user",
            content: `Please summarize these notes:\n\n${noteContent}`
          }
        ]
      };
    }
  );
}
```

github

Common pitfalls

1. **Resource URI conflicts** - Use unique URI schemes
2. **Large resource payloads** - Implement pagination for large datasets
3. **Prompt token limits** - Truncate or summarize content before including in prompts
4. **Missing MIME types** - Always specify appropriate MIME types for resources

Section 5: Error handling and validation

Difficulty: Medium

Estimated Time: 4-5 hours

Purpose: Implement comprehensive error handling and security

Structured error handling

Step 1: Create error middleware (1.5 hours)

typescript

```typescript
export async function withErrorHandling<T>(
  operation: () => Promise<T>,
  context: string
): Promise<{ success: true; data: T } | { success: false; error: string }> {
  try {
    const result = await operation();
    return { success: true, data: result };
  } catch (error) {
    console.error(`Error in ${context}:`, error);

    if (error instanceof NotesError) {
      return {
        success: false,
        error: `${error.code}: ${error.message}`
      };
    }

    if (error instanceof z.ZodError) {
      return {
        success: false,
        error: `Validation failed: ${error.issues[0].message}`
      };
    }

    return {
      success: false,
      error: 'An unexpected error occurred'
    };
  }
}
```

Step 2: Implement security validation (2 hours)

typescript

```
export class SecurityValidator {
  private static readonly DANGEROUS_PATTERNS = [
    /<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi,
    /javascript:/gi,
    /on\w+\s*=/gi
  ];

  static sanitizeInput(input: string): string {
    let sanitized = input;

    for (const pattern of this.DANGEROUS_PATTERNS) {
      sanitized = sanitized.replace(pattern, '');
    }

    return sanitized.slice(0, 10000); // Limit length
  }

  static validatePath(path: string): boolean {
    // Prevent directory traversal
    if (path.includes('..') || path.includes('~')) {
      throw new NotesError('SECURITY_ERROR', 'Invalid path');
    }
    return true;
  }
}
```

redhat

Testing validation

Create comprehensive tests for error scenarios:

```
typescript
```

```
describe('Error Handling', () => {
  it('should handle validation errors gracefully', async () => {
    const result = await server.callTool('create_note', {
      title: '', // Invalid - empty title
      content: 'Test'
    });

    expect(result.isError).toBe(true);
    expect(result.content[0].text).toContain('Validation');
  });
});
```

Section 6: Server lifecycle and transport

Difficulty: Hard

Estimated Time: 6-7 hours

Purpose: Implement proper server lifecycle management and multiple transports

STDIO transport implementation

Step 1: Configure STDIO transport (2 hours)

Update `src/index.ts`: `modelcontextprotocol` `Fka`

typescript

```
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/stdio.js";
import { NotesServer } from "./server.js";

async function startStdioServer() {
  const server = new NotesServer();
  await server.initialize();

  const transport = new StdioServerTransport();

  // Handle graceful shutdown
  process.on('SIGINT', async () => {
    console.error('Shutting down server...');
    await server.shutdown();
    process.exit(0);
  });

  process.on('SIGTERM', async () => {
    console.error('Terminating server...');
    await server.shutdown();
    process.exit(0);
  });

  await server.getServer().connect(transport);
  console.error('Notes MCP server running on STDIO');
}

startStdioServer().catch(error => {
  console.error('Failed to start server:', error);
  process.exit(1);
});
```

github

Step 2: Add HTTP transport option (3 hours)

Create src/transports/http.ts: MCP Framework

typescript

```typescript
import express from 'express';
import { createServer } from 'http';

export class HttpServerTransport {
  private app: express.Application;
  private server: any;

  constructor(private port: number = 3000) {
    this.app = express();
    this.setupMiddleware();
  }

  private setupMiddleware(): void {
    this.app.use(express.json());

    // CORS for browser-based clients
    this.app.use((req, res, next) => {
      res.header('Access-Control-Allow-Origin', '*');
      res.header('Access-Control-Allow-Methods', 'POST, GET, OPTIONS');
      res.header('Access-Control-Allow-Headers', 'Content-Type');
      next();
    });
  }

  async start(mcpServer: any): Promise<void> {
    this.app.post('/mcp', async (req, res) => {
      try {
        const result = await mcpServer.handleRequest(req.body);
        res.json(result);
      } catch (error) {
        res.status(500).json({
          error: 'Internal server error'
        });
      }
    });

    this.server = createServer(this.app);

    return new Promise((resolve) => {
      this.server.listen(this.port, () => {
        console.log(`HTTP server listening on port ${this.port}`);
        resolve();
      });
```

```
    });
  }

  async stop(): Promise<void> {
    return new Promise((resolve) => {
      this.server.close(() => resolve());
    });
  }
}
```

Configuration management

Step 3: Create configuration system (2 hours)

typescript

```typescript
// src/config/settings.ts
import { z } from "zod";

const ConfigSchema = z.object({
  transport: z.enum(['stdio', 'http']).default('stdio'),
  port: z.number().default(3000),
  dataPath: z.string().default('./data/notes.json'),
  maxNotes: z.number().default(10000),
  enableDebug: z.boolean().default(false)
});

export type Config = z.infer<typeof ConfigSchema>;

export function loadConfig(): Config {
  const config = {
    transport: process.env.MCP_TRANSPORT || 'stdio',
    port: parseInt(process.env.MCP_PORT || '3000'),
    dataPath: process.env.MCP_DATA_PATH || './data/notes.json',
    maxNotes: parseInt(process.env.MCP_MAX_NOTES || '10000'),
    enableDebug: process.env.MCP_DEBUG === 'true'
  };

  return ConfigSchema.parse(config);
}
```

Section 7: Testing and validation

Difficulty: Medium

Estimated Time: 5-6 hours

Purpose: Implement comprehensive testing strategy

Unit testing setup

Step 1: Configure Jest (1 hour)

Create `jest.config.js`:

```javascript
export default {
  preset: 'ts-jest',
  testEnvironment: 'node',
  roots: ['<rootDir>/tests'],
  testMatch: ['**/*.test.ts'],
  collectCoverageFrom: [
    'src/**/*.ts',
    '!src/**/*.d.ts',
    '!src/index.ts'
  ],
  coverageThreshold: {
    global: {
      branches: 80,
      functions: 80,
      lines: 80,
      statements: 80
    }
  }
};
```

Step 2: Write comprehensive tests (3 hours)

```typescript
```

```typescript
// tests/tools.test.ts
import { NotesServer } from '../src/server';
import { beforeEach, describe, expect, it } from '@jest/globals';

describe('Note Tools', () => {
  let server: NotesServer;

  beforeEach(async () => {
    server = new NotesServer();
    await server.initialize();
  });

  describe('create_note', () => {
    it('should create note with valid input', async () => {
      const result = await server.callTool('create_note', {
        title: 'Test Note',
        content: 'Test content',
        tags: ['test']
      });

      expect(result.success).toBe(true);
      expect(result.data).toHaveProperty('noteId');
    });

    it('should reject empty title', async () => {
      const result = await server.callTool('create_note', {
        title: '',
        content: 'Test content'
      });

      expect(result.success).toBe(false);
      expect(result.error).toContain('Validation');
    });
  });
});
```

Integration testing

Step 3: Test with MCP Inspector (2 hours)

Create test scenarios document: GitHub Stainless

```markdown
```

```
## MCP Inspector Test Scenarios

### Basic CRUD Operations
1. Create note with title "Meeting Notes"
2. List all notes
3. Get specific note by ID
4. Update note content
5. Delete note

### Error Scenarios
1. Create note with 201+ character title
2. Get non-existent note
3. Delete already deleted note
4. Create note with malicious content

### Performance Tests
1. Create 100 notes rapidly
2. List notes with 1000+ entries
3. Search through large dataset
```

Section 8: Advanced patterns and optimization

Difficulty: Hard

Estimated Time: 8-10 hours

Purpose: Implement production-ready patterns and performance optimization MarkTechPost marktechpost

Implementing caching layer

Step 1: Add Redis caching (3 hours)

```typescript
typescript
```

```
import Redis from 'ioredis';

export class CachedStorage {
  private redis: Redis;
  private storage: JsonNotesStorage;
  private cacheTTL = 300; // 5 minutes

  constructor(storage: JsonNotesStorage) {
    this.storage = storage;
    this.redis = new Redis({
      host: process.env.REDIS_HOST || 'localhost',
      port: parseInt(process.env.REDIS_PORT || '6379')
    });
  }

  async getNote(id: string): Promise<Note | null> {
    // Try cache first
    const cached = await this.redis.get(`note:${id}`);
    if (cached) {
      return JSON.parse(cached);
    }

    // Fetch from storage
    const note = await this.storage.getNote(id);
    if (note) {
      await this.redis.setex(
        `note:${id}`,
        this.cacheTTL,
        JSON.stringify(note)
      );
    }

    return note;
  }

  async invalidateCache(id: string): Promise<void> {
    await this.redis.del(`note:${id}`);
  }
}
```

Implementing streaming responses

Step 2: Add streaming for large datasets (3 hours)

typescript

```typescript
import { Readable } from 'stream';

export class StreamingTools {
  async* streamNotes(filter?: (note: Note) => boolean): AsyncGenerator<Note> {
    const notes = await this.storage.getAllNotes();

    for (const note of notes) {
      if (!filter || filter(note)) {
        yield note;
      }
    }
  }

  createNotesStream(): Readable {
    const stream = new Readable({
      objectMode: true,
      async read() {
        for await (const note of this.streamNotes()) {
          this.push(JSON.stringify(note) + '\n');
        }
        this.push(null);
      }
    });

    return stream;
  }
}
```

BytePlus

Performance monitoring

Step 3: Add metrics collection (2 hours)

typescript

```typescript
export class MetricsCollector {
  private metrics: Map<string, number[]> = new Map();

  recordLatency(operation: string, duration: number): void {
    if (!this.metrics.has(operation)) {
      this.metrics.set(operation, []);
    }

    const values = this.metrics.get(operation)!;
    values.push( duration);

    // Keep only last 1000 measurements
    if (values.length > 1000) {
      values.shift();
    }
  }

  getStats(operation: string): {
    count: number;
    avg: number;
    p95: number;
    p99: number;
  } {
    const values = this.metrics.get(operation) || [];
    if (values.length === 0) {
      return { count: 0, avg: 0, p95: 0, p99: 0 };
    }

    const sorted = [...values].sort((a, b) => a - b);
    const avg = values.reduce((a, b) => a + b, 0) / values.length;
    const p95 = sorted[Math.floor(sorted.length * 0.95)];
    const p99 = sorted[Math.floor(sorted.length * 0.99)];

    return { count: values.length, avg, p95, p99 };
  }
}
```

Section 9: Production deployment

Difficulty: Hard

Estimated Time: 6-8 hours

Purpose: Deploy server for production use

Docker containerization

Step 1: Create Dockerfile (2 hours)

dockerfile

```
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY tsconfig.json ./
COPY src ./src
RUN npm run build

FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY --from=builder /app/build ./build
RUN mkdir -p /app/data

EXPOSE 3000
USER node
CMD ["node", "build/index.js"]
```

Claude Desktop integration

Step 2: Configure for Claude Desktop (2 hours)

Create installation script `install.sh`:

bash

```bash
#!/bin/bash

# Build the server
npm run build

# Get Claude Desktop config path
if [[ "$OSTYPE" == "darwin"* ]]; then
    CONFIG_PATH="$HOME/Library/Application Support/Claude/claude_desktop_config.json"
elif [[ "$OSTYPE" == "msys" ]] || [[ "$OSTYPE" == "cygwin" ]]; then
    CONFIG_PATH="$APPDATA/Claude/claude_desktop_config.json"
else
    CONFIG_PATH="$HOME/.config/Claude/claude_desktop_config.json"
fi

# Create config if it doesn't exist
if [ ! -f "$CONFIG_PATH" ]; then
    echo '{"mcpServers": {}}' > "$CONFIG_PATH"
fi

# Add server configuration
cat <<EOF > /tmp/notes_server_config.json
{
  "notes": {
    "command": "node",
    "args": ["$(pwd)/build/index.js"],
    "env": {
      "MCP_DATA_PATH": "$(pwd)/data/notes.json"
    }
  }
}
EOF

# Merge configuration
jq '.mcpServers += input' "$CONFIG_PATH" /tmp/notes_server_config.json > /tmp/merged_config.json
mv /tmp/merged_config.json "$CONFIG_PATH"

echo "Notes MCP server installed successfully!"
echo "Restart Claude Desktop to use the server."
```

Production monitoring

Step 3: Set up monitoring (2 hours)

typescript

```typescript
// src/monitoring/health.ts
export class HealthCheck {
  private lastCheck: Date = new Date();
  private isHealthy: boolean = true;

  async checkHealth(): Promise<{
    status: 'healthy' | 'unhealthy';
    uptime: number;
    version: string;
    checks: Record<string, boolean>;
  }> {
    const checks = {
      storage: await this.checkStorage(),
      memory: this.checkMemory(),
      diskSpace: await this.checkDiskSpace()
    };

    const allHealthy = Object.values(checks).every(v => v);

    return {
      status: allHealthy ? 'healthy' : 'unhealthy',
      uptime: process.uptime(),
      version: '1.0.0',
      checks
    };
  }

  private async checkStorage(): Promise<boolean> {
    try {
      await this.storage.getAllNotes();
      return true;
    } catch {
      return false;
    }
  }

  private checkMemory(): boolean {
    const used = process.memoryUsage();
    return used.heapUsed < 500 * 1024 * 1024; // 500MB limit
  }
}
```

Section 10: Documentation and maintenance

Difficulty: Easy

Estimated Time: 3-4 hours

Purpose: Create comprehensive documentation for users and developers

User documentation

Create `README.md`:

````
markdown

# Notes MCP Server

A Model Context Protocol server for managing personal notes with full CRUD operations.

## Installation

### Quick Install (Claude Desktop)
```bash
npm install
npm run build
npm run install:claude
````

Manual Installation

1. Clone the repository
2. Install dependencies: `npm install`
3. Build the server: `npm run build`
4. Add to Claude Desktop configuration

Available Tools

create_note

Create a new note with title and content.

Parameters:

- `title` (string, required): Note title (1-200 characters)
- `content` (string, required): Note content
- `tags` (string[], optional): Tags for categorization
````

list_notes

List all notes with optional filtering.

Parameters:

- `tag` (string, optional): Filter by tag
- `limit` (number, optional): Maximum results (default: 20)

Configuration

Environment variables:

- `MCP_DATA_PATH`: Path to notes storage file
- `MCP_TRANSPORT`: Transport type (stdio|http)
- `MCP_PORT`: HTTP server port (default: 3000)
- `MCP_DEBUG`: Enable debug logging

Developer documentation

Create `DEVELOPMENT.md`:
```markdown
# Development Guide

## Architecture

The server follows a modular architecture:
- **Storage Layer**: Handles data persistence
- **Tool Layer**: Implements MCP tool handlers
- **Transport Layer**: Manages client communication
- **Validation Layer**: Ensures data integrity

## Testing

Run tests: `npm test`
Run with coverage: `npm run test:coverage`
Use MCP Inspector: `npm run inspector`

## Common Issues

### STDIO Transport Issues
- Never write to stdout
- Use console.error for logging
- Check Node.js version (18+)

### Storage Issues
- Ensure data directory exists
- Check file permissions
- Validate JSON syntax

## Final checklist and best practices

### Security checklist

- ✅ Input validation on all tools
- ✅ Path traversal prevention
- ✅ HTML/script injection protection
- ✅ Rate limiting for API endpoints
```

- ✅ Proper error messages (no stack traces)
- ✅ Environment variable validation
- ✅ File permission checks

Performance checklist

- ✅ Caching for frequently accessed data
- ✅ Pagination for large datasets
- ✅ Async/await for all I/O operations
- ✅ Connection pooling for external services
- ✅ Memory usage monitoring
- ✅ Graceful shutdown handling

Testing checklist

- ✅ Unit tests with 80% coverage
- ✅ Integration tests for all tools
- ✅ Error scenario testing
- ✅ Performance benchmarks
- ✅ MCP Inspector validation
- ✅ Multi-client testing

Deployment checklist

- ✅ Docker containerization
- ✅ Health check endpoints
- ✅ Logging configuration
- ✅ Monitoring setup
- ✅ Backup strategy
- ✅ Documentation complete

Conclusion

This comprehensive development plan provides a structured path to building a production-ready MCP server. The Personal Note Manager project teaches essential MCP concepts while implementing real-world patterns and best practices. By following this guide, developers gain hands-on experience with

the Model Context Protocol ecosystem, preparing them to build more complex MCP integrations for AI-powered applications.

The modular architecture, comprehensive error handling, and security-first approach ensure the server is not just a learning exercise but a foundation for production deployments. As the MCP ecosystem continues to evolve in 2025, these fundamental patterns and practices will remain relevant for building robust AI tool integrations.