Learning Path for Backend and AI Services

Based on your project requirements, here's a detailed learning path for implementing the backend (Node.js/Express) and AI service (Python/FastAPI) components:

**Phase 1**: Node.js Backend Fundamentals (5-7 days)

Core Concepts to Learn:

- Express.js framework and middleware
- REST API design principles
- Environment variables and configuration
- Error handling and logging
- Project structure and organization

Key Implementation:

javascript

```
// Basic Express server setup
const express = require('express');
const app = express();
app.use(express.json());

// Routes
app.use('/api/auth', require('./routes/auth'));
app.use('/api/reviews', require('./routes/reviews'));
app.use('/webhooks/github', require('./routes/webhooks'));

// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Something went wrong!' });
});
```

**Phase 2**: Database Integration with MongoDB (3-5 days)

Concepts to Learn:

- MongoDB basics and Mongoose ODM
- Schema design for users, PR metadata, and review results

- CRUD operations
- Data validation
- Indexing for performance

Implementation Code:

javascript

```
// User schema
const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  githubToken: String,
  createdAt: { type: Date, default: Date.now }
});

// Review schema
const reviewSchema = new mongoose.Schema({
  prId: { type: String, required: true },
  repo: { type: String, required: true },
  prTitle: { type: String, required: true },
  score: Number,
  categories: {
    lint: Number,
    bugs: Number,
    security: Number,
    performance: Number
  },
  summary: String,
  comments: [{
    path: String,
    line: Number,
    body: String
  }],
  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  createdAt: { type: Date, default: Date.now }
});
```

**Phase 3**: Authentication Implementation (3-5 days)

Concepts to Learn:

- JWT (JSON Web Tokens) authentication flow
- Password hashing with bcrypt
- Protected routes middleware
- Token refresh strategies

Implementation Code:

javascript

```
// Authentication middleware
const jwt = require('jsonwebtoken');

const authenticateToken = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];

  if (!token) {
    return res.status(401).json({ message: 'Access token required' });
  }

  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) {
      return res.status(403).json({ message: 'Invalid token' });
    }
    req.user = user;
    next();
  });
};
```

```
// Password hashing
const bcrypt = require('bcrypt');
const saltRounds = 10;

const hashPassword = async (password) => {
  return await bcrypt.hash(password, saltRounds);
};

const comparePassword = async (password, hash) => {
  return await bcrypt.compare(password, hash);
};
```

**Phase 4:** Webhook Handling and GitHub API (5-7 days)

Concepts to Learn:

- GitHub webhook security with HMAC verification
- GitHub REST API integration
- Asynchronous processing of webhook events
- Error handling for external API calls

Implementation Code:

javascript

```
// Webhook signature verification
const crypto = require('crypto');

const verifyGitHubSignature = (req, res, next) => {
  const signature = req.headers['x-hub-signature-256'];
  const hmac = crypto.createHmac('sha256', process.env.GITHUB_WEBHOOK_SECRET);
  const digest = 'sha256=' + hmac.update(JSON.stringify(req.body)).digest('hex');

  if (signature === digest) {
    next();
  } else {
    res.status(401).send('Invalid signature');
  }
};

// GitHub API service
const axios = require('axios');

class GitHubService {
  constructor() {
    this.baseURL = 'https://api.github.com';
    this.headers = {
      'Authorization': `token ${process.env.GITHUB_ACCESS_TOKEN}`,
      'Accept': 'application/vnd.github.v3+json',
      'User-Agent': 'AI-Code-Reviewer'
    };
  }

  async getPRFiles(owner, repo, pullNumber) {
    const response = await axios.get(
      `${this.baseURL}/repos/${owner}/${repo}/pulls/${pullNumber}/files`,
      { headers: this.headers }
    );
    return response.data;
  }

  async postReviewComment(owner, repo, pullNumber, reviewData) {
    const response = await axios.post(
      `${this.baseURL}/repos/${owner}/${repo}/pulls/${pullNumber}/reviews`,
      reviewData,
      { headers: this.headers }
    );
    return response.data;
  }
}
```

**Phase 5:** Python FastAPI Service (5-7 days)

Concepts to Learn:

- FastAPI framework and endpoints
- Pydantic models for request/response validation
- CORS middleware for cross-origin requests
- API documentation with Swagger/OpenAPI

Implementation Code:

python

```
# FastAPI app setup
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import List, Optional

app = FastAPI(title="AI Code Review Service")

# CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],  # React app URL
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Pydantic models
class CodeFile(BaseModel):
    path: str
    content: str
    changes: str

class AnalysisRequest(BaseModel):
    pr_id: str
    repo: str
    files: List[CodeFile]

class Comment(BaseModel):
    path: str
    line: int
    body: str

class FixSuggestion(BaseModel):
    path: str
    patch: str

class AnalysisResponse(BaseModel):
    score: int
    categories: dict
    summary: str
    comments: List[Comment]
    fix_suggestions: List[FixSuggestion]

# Analysis endpoint
@app.post("/analyze", response_model=AnalysisResponse)
async def analyze_code(request: AnalysisRequest):
    try:
        # Process each file with AI agents
        analysis_results = await process_with_ai_agents(request.files)
        return analysis_results
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

**Phase 6:** AI Service with LangChain and LangGraph (7-10 days)

Concepts to Learn:

- LangChain framework for AI workflows
- LangGraph for multi-agent orchestration
- LLM API integration (OpenAI, Anthropic, or others)
- Prompt engineering for code analysis
- Structured output parsing

Implementation Code:

python

```
# AI Agent setup
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage, SystemMessage
from langgraph.graph import Graph
from typing import Dict, Any
import json

# Initialize LLM
llm = ChatOpenAI(
    model="gpt-4",
    temperature=0,
    openai_api_key=os.getenv("OPENAI_API_KEY")
)

# Agent definitions
class LintStyleAgent:
    def __init__(self, llm):
        self.llm = llm

    async def analyze(self, code: str) -> Dict[str, Any]:
        system_prompt = """You are a code quality expert. Analyze the code for:
        1. Code formatting and style issues
        2. Best practices violations
        3. Readability improvements
        Return JSON with score (0-100) and specific comments."""

        response = await self.llm.agenerate([
            [
                SystemMessage(content=system_prompt),
                HumanMessage(content=code)
            ]
        ])
        return json.loads(response.generations[0][0].text)

# Similar agents for BugDetector, SecurityScanner, PerformanceReviewer

# Coordinator agent
class CoordinatorAgent:
    def __init__(self, agents):
        self.agents = agents

    async def coordinate_analysis(self, files):
        results = {}
        for agent_name, agent in self.agents.items():
            agent_results = []
            for file in files:
                analysis = await agent.analyze(file.content)
                agent_results.append(analysis)
            results[agent_name] = agent_results

        # Merge results and create final output
        return self.merge_results(results)

    def merge_results(self, results):
        # Implementation to merge results from all agents
        pass

# LangGraph workflow
workflow = Graph()

workflow.add_node("lint_analysis", LintStyleAgent(llm).analyze)
workflow.add_node("bug_analysis", BugDetector(llm).analyze)
workflow.add_node("security_analysis", SecurityScanner(llm).analyze)
workflow.add_node("performance_analysis", PerformanceReviewer(llm).analyze)
workflow.add_node("coordinator", CoordinatorAgent().merge_results)

# Define edges
workflow.add_edge("lint_analysis", "coordinator")
workflow.add_edge("bug_analysis", "coordinator")
workflow.add_edge("security_analysis", "coordinator")
workflow.add_edge("performance_analysis", "coordinator")
```

Phase 7: Service Communication (3-5 days)

Concepts to Learn:

- HTTP communication between Node.js and Python services
- Error handling and retry mechanisms
- Data serialization/deserialization
- Timeout management

Implementation Code:

javascript

```
// Node.js service to call Python AI service
const axios = require('axios');
```

```
class AIServiceClient {
  constructor(baseURL) {
    this.client = axios.create({
      baseURL: baseURL || process.env.AI_SERVICE_URL,
      timeout: 30000,
      headers: {
        'Content-Type': 'application/json'
      }
    });
  }

  async analyzeCode(prData, files) {
    try {
      const payload = {
        pr_id: prData.prId,
        repo: prData.repo,
        files: files.map(file => ({
          path: file.filename,
          content: file.patch, // Or fetch full content if needed
          changes: file.patch
        }))
      };

      const response = await this.client.post('/analyze', payload);
      return response.data;
    } catch (error) {
      console.error('Error calling AI service:', error.message);
      throw new Error('Failed to analyze code');
    }
  }
}
```

**Phase 8:** Error Handling and Logging (3-5 days)

Concepts to Learn:

- Comprehensive error handling strategies
- Structured logging
- Monitoring and alerting
- Retry mechanisms for external services

Implementation Code:

javascript

```
// Enhanced error handling with retries
const retry = require('async-retry');

async function processPREventWithRetry(payload) {
  return await retry(
    async (bail, attempt) => {
      try {
        console.log(`Processing PR event, attempt ${attempt}`);
        const result = await processPREvent(payload);
        return result;
      } catch (error) {
        if (error.isRetryable === false) {
          bail(error); // Don't retry if not retryable
        }
        throw error; // Retry for other errors
      }
    },
    {
      retries: 3,
      minTimeout: 1000,
      maxTimeout: 5000
    }
  );
}

// Structured logging
const winston = require('winston');

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json(),
  transports: [
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' })
  ]
});

if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console({
    format: winston.format.simple()
  }));
}
```

Recommended Resources:

1. Express.js documentation
2. MongoDB University free courses
3. FastAPI documentation
4. LngChain documentation
5. GitHub REST API documentation
6. JWT.io for token debugging

Timeline Estimate:

- Total: 30-45 days (learning while building)
- Accelerated pace: 4-5 weeks with full-time focus

This learning path will give you a solid foundation to implement both the backend and AI service components of your AI-powered code review system. Remember to build and test each component incrementally, starting with the basic structure and gradually adding complexity.