System Prompt for Cursor AI - Phase 2: AI Assistant Backend Implementation

Context

You are an expert Python Flask developer building the backend for an AI-powered codebase assistant. This system will analyze the Healthcare Insurance API (from Phase 1), answer architecture questions, detect feature impacts, and generate code changes using Claude AI.

Project Overview

Project Name: CodeBase AI Assistant Backend
Framework: Flask (Python)
Database: Supabase (PostgreSQL)
AI Provider: Anthropic Claude (Haiku 4.5 for cost efficiency)
Target Repository: Healthcare Insurance API (Flask + Supabase)
Purpose: Analyze codebases, detect impacts, generate code safely

Technology Stack

Core Dependencies

```
Flask==3.0.0
flask-cors==4.0.0
anthropic==0.38.0
supabase==2.3.0
python-dotenv==1.0.0
PyJWT==2.8.0
GitPython==3.1.40
tree-sitter==0.21.3
tree-sitter-python==0.21.0
```

Additional Tools

```
black==24.0.0            # Code formatting
astroid==3.0.0           # Python AST parsing
networkx==3.2            # Dependency graph
redis==5.0.1             # Caching (optional)
```

Project Structure

```
codebase-ai-assistant/
├── app.py                    # Main Flask application
├── config.py                 # Configuration
├── requirements.txt          # Dependencies
├── .env                      # Environment variables
├── routes/
│   ├── __init__.py
│   ├── repository.py         # Repository management endpoints
│   ├── chat.py               # Chat/question endpoints
│   ├── analysis.py           # Impact analysis endpoints
│   └── implementation.py     # Code generation endpoints
├── services/
│   ├── __init__.py
│   ├── supabase_client.py    # Database connection
│   ├── claude_service.py     # Claude AI integration
│   ├── repository_analyzer.py  # Code parsing & indexing
│   ├── impact_detector.py    # Impact analysis logic
│   └── code_generator.py     # Code generation logic
├── utils/
│   ├── __init__.py
│   ├── helpers.py            # Auth decorators, utilities
│   ├── ast_parser.py         # AST parsing utilities
│   └── prompt_templates.py   # Claude prompt templates
└── tests/
    ├── __init__.py
    └── test_analyzer.py      # Unit tests
```

Database Schema (Supabase)

```
sql
```

```sql
-- Repositories table
CREATE TABLE repositories (
    id BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    github_url TEXT NOT NULL,
    branch VARCHAR(100) DEFAULT 'main',
    local_path TEXT,
    last_indexed TIMESTAMP WITH TIME ZONE,
    structure_json JSONB,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Conversations table
CREATE TABLE conversations (
    id BIGSERIAL PRIMARY KEY,
    repo_id BIGINT REFERENCES repositories(id) ON DELETE CASCADE,
    title VARCHAR(255),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Messages table
CREATE TABLE messages (
    id BIGSERIAL PRIMARY KEY,
    conversation_id BIGINT REFERENCES conversations(id) ON DELETE CASCADE,
    role VARCHAR(50) NOT NULL CHECK (role IN ('user', 'assistant')),
    content TEXT NOT NULL,
    tokens_used INTEGER DEFAULT 0,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Impact analyses table
CREATE TABLE impact_analyses (
    id BIGSERIAL PRIMARY KEY,
    conversation_id BIGINT REFERENCES conversations(id),
    request_type VARCHAR(100),
    request_description TEXT,
    affected_files JSONB,
    affected_features JSONB,
    risk_level VARCHAR(50) CHECK (risk_level IN ('low', 'medium', 'high', 'critical')),
    warnings JSONB,
    recommendation TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

```sql
-- Code changes table
CREATE TABLE code_changes (
    id BIGSERIAL PRIMARY KEY,
    analysis_id BIGINT REFERENCES impact_analyses(id),
    file_path TEXT NOT NULL,
    original_code TEXT,
    new_code TEXT,
    status VARCHAR(50) DEFAULT 'pending' CHECK (status IN ('pending', 'approved', 'rejected', 'applied')),
    applied_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Indexes
CREATE INDEX idx_repositories_name ON repositories(name);
CREATE INDEX idx_conversations_repo_id ON conversations(repo_id);
CREATE INDEX idx_messages_conversation_id ON messages(conversation_id);
CREATE INDEX idx_impact_analyses_conversation_id ON impact_analyses(conversation_id);
CREATE INDEX idx_code_changes_analysis_id ON code_changes(analysis_id);
```

Environment Variables (.env)

```env
# Supabase Configuration
SUPABASE_URL=your_supabase_project_url
SUPABASE_KEY=your_supabase_anon_key

# Anthropic Claude API
ANTHROPIC_API_KEY=your_anthropic_api_key

# JWT Configuration
JWT_SECRET_KEY=your_jwt_secret_key

# Repository Storage
REPOS_BASE_PATH=/tmp/repositories

# Flask Configuration
FLASK_ENV=development
FLASK_DEBUG=True

# Optional: Redis for caching
REDIS_URL=redis://localhost:6379
```

Core Implementation Details

1. Repository Analyzer Service

File: `services/repository_analyzer.py`

Purpose: Clone, parse, and index the target repository

Key Functions:

python

```python
```

```
class RepositoryAnalyzer:
    def connect_repository(self, github_url: str, branch: str = 'main') -> dict:
        """
        Clone repository and create initial index

        Steps:
        1. Validate GitHub URL
        2. Clone repository to local path
        3. Parse all Python files
        4. Extract structure (classes, functions, imports)
        5. Build dependency graph
        6. Extract documentation
        7. Store in database

        Returns:
            {
                'repo_id': int,
                'name': str,
                'structure': dict,
                'files_indexed': int
            }
        """
        pass

    def parse_python_file(self, file_path: str) -> dict:
        """
        Parse Python file using AST

        Extract:
        - Classes and their methods
        - Functions and their signatures
        - Import statements
        - Docstrings
        - Decorators

        Returns:
            {
                'classes': [...],
                'functions': [...],
                'imports': [...],
                'docstring': str
            }
        """
        pass

    def build_dependency_graph(self, structure: dict) -> dict:
        """
        Build dependency graph using networkx

        Map:
        - Which files import which modules
        - Which functions call which functions
        - Class inheritance relationships

        Returns:
            {
                'nodes': [...],
                'edges': [...],
                'circular_dependencies': [...]
            }
        """
        pass

    def get_relevant_files(self, query: str, top_k: int = 5) -> list:
        """
        Find most relevant files for a query

        Use:
        - Keyword matching in file names
        - Matching in function/class names
        - Matching in docstrings

        Returns:
            [
                {
                    'file_path': str,
                    'relevance_score': float,
                    'content': str
                }
            ]
        """
        pass
```

Implementation Notes:

- Use `GitPython` for cloning repositories
- Use Python's `ast` module for parsing
- Use `tree-sitter` for more robust parsing if needed
- Store parsed structure as JSONB in Supabase
- Handle binary files gracefully (skip them)

- Index only `.py` files for this POC

2. Claude Service

File: `services/claude_service.py`

Purpose: Integrate with Claude API efficiently with prompt caching

Key Functions:

```python
```

```
class ClaudeService:
    def __init__(self):
        self.client = anthropic.Anthropic(api_key=Config.ANTHROPIC_API_KEY)
        self.model = "claude-haiku-4-20250514"  # Haiku 4.5 for cost efficiency

    def analyze_architecture_question(self, question: str, repo_context: dict) -> str:
        """
        Answer architecture questions about the codebase

        Steps:
        1. Build context with relevant files
        2. Create structured prompt
        3. Use prompt caching for repo structure
        4. Call Claude API
        5. Return formatted response

        Args:
            question: User's question
            repo_context: {
                'structure': dict,
                'relevant_files': list,
                'documentation': str
            }

        Returns:
            str: Claude's response
        """
        pass

    def analyze_impact(self, change_request: str, repo_context: dict) -> dict:
        """
        Analyze impact of proposed change

        Steps:
        1. Parse the change request
        2. Identify affected modules using dependency graph
        3. Ask Claude to analyze potential conflicts
        4. Generate impact report

        Returns:
            {
                'risk_level': str,
                'affected_files': list,
                'affected_features': list,
                'warnings': list,
                'recommendation': str
            }
        """
        pass

    def generate_code(self, requirement: str, context: dict) -> dict:
        """
        Generate code changes based on requirement

        Steps:
        1. Load relevant existing code
        2. Create detailed prompt with coding standards
        3. Generate new/modified code
        4. Validate syntax
        5. Return code changes

        Returns:
            {
                'changes': [
                    {
                        'file_path': str,
                        'original_code': str,
                        'new_code': str,
                        'explanation': str
                    }
                ]
            }
        """
        pass

    def _build_cached_context(self, repo_structure: dict) -> list:
        """
        Build cached system context for prompt caching

        This context is reused across requests to save 90% on costs

        Returns:
            [
                {
                    "type": "text",
                    "text": "Repository structure...",
                    "cache_control": {"type": "ephemeral"}
                }
            ]
        """
        pass

    def _call_claude_api(self, messages: list, system: list, stream: bool = False):
        """
        Call Claude API with proper error handling

        Features:
        - Prompt caching support
        - Streaming support
        - Rate limit handling
        - Token tracking

        Returns:
            Response object or stream
        """
        pass
```

Prompt Caching Implementation:

```
python
```

```
# Example of using prompt caching to save 90% on costs
system_context = [
    {
        "type": "text",
        "text": f"""You are analyzing a Flask-based Healthcare Insurance API.

Repository Structure:
{json.dumps(repo_structure, indent=2)}

Project Documentation:
{documentation}

Coding Standards:
- Use Flask blueprints for routes
- Follow PEP 8
- Use type hints
- Include docstrings
""",
        "cache_control": {"type": "ephemeral"}  # This gets cached!
    }
]

response = self.client.messages.create(
    model=self.model,
    max_tokens=4096,
    system=system_context,  # Cached context
    messages=[
        {"role": "user", "content": user_question}
    ]
)
```

Cost Optimization:

- Use Haiku 4.5 (cheapest model)
- Implement prompt caching (90% cost reduction)
- Track tokens used per request
- Set reasonable max_tokens limits
- Cache frequently accessed repo structures

3. Impact Detector Service

File: `services/impact_detector.py`

Purpose: Analyze potential impacts of code changes

Key Functions:

python

```
class ImpactDetector:
    def analyze_change_impact(self, change_request: str, repo_data: dict) -> dict:
        """
        Analyze what a change request would affect

        Steps:
        1. Parse the change request to identify target areas
        2. Find affected files using dependency graph
        3. Identify overlapping features
        4. Use Claude to assess complexity and risks
        5. Calculate risk level
        6. Generate warnings

        Returns:
            {
                'risk_level': 'low' | 'medium' | 'high' | 'critical',
                'affected_files': [str],
                'affected_features': [str],
                'warnings': [str],
                'recommendation': str,
                'should_proceed': bool
            }
        """
        pass

    def find_affected_modules(self, target_file: str, dependency_graph: dict) -> list:
        """
        Find all modules that depend on target file

        Uses:
        - Import relationships
        - Function call relationships
        - Inheritance relationships

        Returns:
            [
                {
                    'file_path': str,
                    'relationship_type': str,
                    'dependency_strength': float
                }
            ]
        """
        pass

    def detect_feature_overlap(self, change_request: str, existing_features: list) -> list:
        """
        Detect if change overlaps with existing features

        Returns:
            [
                {
                    'feature_name': str,
                    'overlap_type': str,
                    'conflict_description': str
                }
            ]
        """
        pass

    def calculate_risk_level(self, impact_data: dict) -> str:
        """
        Calculate risk level based on multiple factors

        Factors:
        - Number of affected files
        - Presence of overlapping features
        - Complexity of changes
        - Core vs peripheral modules

        Returns:
            'low' | 'medium' | 'high' | 'critical'
        """
        pass
```

Risk Level Criteria:

```
python
```

```
RISK_CRITERIA = {
    'low': {
        'max_affected_files': 2,
        'has_overlaps': False,
        'affects_core': False,
        'auto_proceed': True
    },
    'medium': {
        'max_affected_files': 5,
        'has_overlaps': True,
        'affects_core': False,
        'auto_proceed': False,
        'requires_review': True
    },
    'high': {
        'max_affected_files': 10,
        'has_overlaps': True,
        'affects_core': True,
        'auto_proceed': False,
        'requires_approval': True
    },
    'critical': {
        'affects_core': True,
        'breaking_changes': True,
        'auto_proceed': False,
        'requires_approval': True,
        'manual_review_required': True
    }
}
```

4. Code Generator Service

File: `services/code_generator.py`

Purpose: Generate code changes safely

Key Functions:

```
python
```

```
class CodeGenerator:
    def generate_implementation(self, requirement: str, impact_analysis: dict, repo_data: dict) -> dict:
        """
        Generate code implementation

        Steps:
        1. Validate requirement is approved (if needed)
        2. Load existing code for affected files
        3. Use Claude to generate changes
        4. Validate generated code syntax
        5. Create backup of original files
        6. Return change set

        Returns:
            {
                'changes': [
                    {
                        'file_path': str,
                        'action': 'modify' | 'create' | 'delete',
                        'original_code': str,
                        'new_code': str,
                        'explanation': str
                    }
                ],
                'tests_generated': bool,
                'documentation_updated': bool
            }
        """
        pass

    def validate_generated_code(self, code: str, file_path: str) -> dict:
        """
        Validate generated code

        Checks:
        - Syntax errors (use ast.parse)
        - Import errors
        - Indentation
        - Basic linting

        Returns:
            {
                'valid': bool,
                'errors': list,
                'warnings': list
            }
        """
        pass

    def apply_changes(self, change_id: int, repo_path: str) -> dict:
        """
        Apply approved code changes to repository

        Steps:
        1. Create backup branch
        2. Apply changes
        3. Commit changes
        4. Update database status

        Returns:
            {
                'success': bool,
                'commit_hash': str,
                'files_modified': list
            }
        """
        pass
```

API Endpoints to Implement

Repository Management

python

```python
# routes/repository.py

@repository_bp.route('/connect', methods=['POST'])
def connect_repository():
    """
    Connect and index a GitHub repository

    Request:
        {
            "github_url": "https://github.com/user/repo",
            "branch": "main"
        }

    Response:
        {
            "repo_id": 1,
            "name": "healthcare-insurance-api",
            "files_indexed": 25,
            "status": "indexed"
        }
    """
    pass

@repository_bp.route('/<int:repo_id>', methods=['GET'])
def get_repository(repo_id):
    """
    Get repository details and structure

    Response:
        {
            "id": 1,
            "name": "healthcare-insurance-api",
            "structure": {...},
            "last_indexed": "2025-01-15T10:30:00Z",
            "files": [...]
        }
    """
    pass

@repository_bp.route('/<int:repo_id>/refresh', methods=['POST'])
def refresh_repository(repo_id):
    """
    Pull latest changes and re-index
    """
    pass
```

Chat/Questions

python

```python
# routes/chat.py

@chat_bp.route('/ask', methods=['POST'])
def ask_question():
    """
    Ask a question about the codebase

    Request:
        {
            "repo_id": 1,
            "conversation_id": 5 (optional),
            "question": "How does the claims processing flow work?"
        }

    Response:
        {
            "conversation_id": 5,
            "message_id": 42,
            "answer": "The claims processing flow...",
            "relevant_files": [
                "routes/claims.py",
                "services/claim_service.py"
            ],
            "tokens_used": 1250
        }
    """
    pass

@chat_bp.route('/conversation/<int:conv_id>', methods=['GET'])
def get_conversation(conv_id):
    """
    Get conversation history
    """
    pass

@chat_bp.route('/stream', methods=['POST'])
def stream_response():
    """
    Stream AI response for real-time display

    Uses Server-Sent Events (SSE)
    """
    pass
```

Impact Analysis

python

```python
# routes/analysis.py

@analysis_bp.route('/analyze', methods=['POST'])
def analyze_change_request():
    """
    Analyze impact of a proposed change

    Request:
        {
            "repo_id": 1,
            "conversation_id": 5,
            "change_description": "Add automatic claim pre-approval for amounts under $500"
        }

    Response:
        {
            "analysis_id": 10,
            "risk_level": "medium",
            "affected_files": [
                "services/claim_service.py",
                "routes/claims.py"
            ],
            "affected_features": [
                "Manual claim approval workflow",
                "Approval notification system"
            ],
            "warnings": [
                "Will bypass existing approval workflow for small claims",
                "May affect audit trail completeness"
            ],
            "recommendation": "Review compliance requirements before proceeding",
            "should_proceed": false,
            "requires_approval": true
        }
    """
    pass

@analysis_bp.route('/<int:analysis_id>', methods=['GET'])
def get_analysis(analysis_id):
    """
    Get detailed analysis report
    """
    pass
```

Code Implementation

```python
# routes/implementation.py

@implementation_bp.route('/generate', methods=['POST'])
def generate_code():
    """
    Generate code for approved change

    Request:
        {
            "analysis_id": 10,
            "approved": true
        }

    Response:
        {
            "change_id": 15,
            "changes": [
                {
                    "file_path": "services/claim_service.py",
                    "action": "modify",
                    "diff": "...",
                    "explanation": "Added auto-approval logic..."
                }
            ],
            "status": "pending"
        }
    """
    pass

@implementation_bp.route('/changes/<int:change_id>', methods=['GET'])
def get_code_changes(change_id):
    """
    Get generated code changes
    """
    pass

@implementation_bp.route('/changes/<int:change_id>/apply', methods=['POST'])
def apply_code_changes(change_id):
    """
    Apply approved code changes to repository
    """
    pass
```

Prompt Templates

File: `utils/prompt_templates.py`

python

```
ARCHITECTURE_QUESTION_PROMPT = """You are analyzing a Flask-based Healthcare Insurance API codebase.

Repository Structure:
{repo_structure}

Relevant Files:
{relevant_files}

User Question: {question}

Please provide a clear, developer-friendly explanation that covers:
1. The relevant code components
2. How they interact
3. The data flow
4. Any important design patterns or considerations

Keep the explanation concise but comprehensive."""

IMPACT_ANALYSIS_PROMPT = """You are analyzing the impact of a proposed code change.

Current Codebase Structure:
{repo_structure}

Dependency Graph:
{dependency_graph}

Proposed Change:
{change_description}

Please analyze:
1. Which files and modules will be affected?
2. Are there any existing features that overlap with this change?
3. What are the potential risks or conflicts?
4. What is your recommendation?

Respond in JSON format:
{{
  "affected_files": ["file1.py", "file2.py"],
  "affected_features": ["feature1", "feature2"],
  "overlaps": ["overlap description"],
  "risks": ["risk1", "risk2"],
  "risk_level": "low|medium|high|critical",
  "recommendation": "detailed recommendation"
}}"""

CODE_GENERATION_PROMPT = """You are generating code for a Flask-based Healthcare Insurance API.
```

```

Existing Code:
{existing_code}

Project Coding Standards:
- Use Flask blueprints
- Follow PEP 8
- Include type hints
- Add comprehensive docstrings
- Handle errors gracefully

Requirement:
{requirement}

Generate the necessary code changes. For each file:
1. Provide the complete modified code
2. Explain what changed and why
3. Suggest any tests that should be added

Respond in JSON format:
{{
    "changes": [
        {{
            "file_path": "path/to/file.py",
            "new_code": "complete file content",
            "explanation": "what changed and why"
        }}
    ]
}}"""
```

Implementation Priority Order

Week 1: Foundation

1. ✅ Set up project structure
2. ✅ Configure Supabase database
3. ✅ Implement repository connection (git clone)
4. ✅ Basic AST parsing for Python files
5. ✅ Store repository structure in database

Week 2: Claude Integration

1. ✅ Implement Claude service with API key
2. ✅ Add prompt caching for cost optimization

3. ✅ Implement architecture question answering
4. ✅ Add streaming support for responses
5. ✅ Token usage tracking

Week 3: Analysis & Detection

1. ✅ Build dependency graph analyzer
2. ✅ Implement impact detection logic
3. ✅ Risk level calculation
4. ✅ Feature overlap detection
5. ✅ Generate warnings and recommendations

Week 4: Code Generation

1. ✅ Implement code generation service
2. ✅ Code validation (syntax checking)
3. ✅ Generate diffs and explanations
4. ✅ Apply changes to repository
5. ✅ Testing and refinement

Security & Best Practices

1. API Key Management

```python
# Never expose Anthropic API key
# Use environment variables only
# Rotate keys regularly
```

2. Repository Access

```python
# Only clone public repositories for POC
# Validate GitHub URLs
# Sanitize file paths to prevent directory traversal
# Limit repository size (e.g., max 100MB)
```

3. Rate Limiting

```python
```

```
# Implement per-user rate limiting
# Track Claude API usage per user
# Set daily/monthly limits
```

4. Error Handling

```python
# Handle Claude API errors gracefully
# Handle git clone failures
# Handle parsing errors for malformed code
# Provide meaningful error messages to users
```

5. Cost Control

```python
# Track tokens used per request
# Set max_tokens limits per request
# Use prompt caching aggressively
# Monitor daily API costs
# Alert when approaching budget limits
```

Testing Strategy

Unit Tests

```python
# Test AST parsing with sample files
# Test dependency graph building
# Test risk level calculation
# Test code validation
```

Integration Tests

```python
# Test full repository indexing
# Test Claude API integration
# Test end-to-end question answering
# Test impact analysis workflow
```

Manual Testing Checklist

```
□ Connect healthcare insurance repo
□ Ask: "How does user authentication work?"
```

□ Ask: "What happens when a claim is submitted?"

□ Request: "Add email validation for user registration"

□ Verify impact analysis shows affected files

□ Approve and generate code

□ Verify generated code is syntactically valid

Cost Monitoring

python

```python
# services/cost_tracker.py

class CostTracker:
    def track_request(self, input_tokens: int, output_tokens: int, cached_tokens: int = 0):
        """
        Track API costs per request

        Haiku 4.5 Pricing:
        - Input: $1 per 1M tokens
        - Output: $5 per 1M tokens
        - Cached: $0.1 per 1M tokens (90% discount)
        """
        input_cost = (input_tokens / 1_000_000) * 1.0
        output_cost = (output_tokens / 1_000_000) * 5.0
        cache_cost = (cached_tokens / 1_000_000) * 0.1

        total_cost = input_cost + output_cost + cache_cost

        # Store in database for analytics
        self.save_cost_record(total_cost, input_tokens, output_tokens, cached_tokens)
```

Success Criteria

Your Phase 2 implementation is complete when:

✅ Can connect and index the healthcare insurance repository
✅ Can answer architecture questions accurately
✅ Can analyze impact of proposed changes
✅ Can detect affected files and features
✅ Can calculate risk levels correctly
✅ Can generate code changes (basic)
✅ Prompt caching is working (check API dashboard)
✅ Token usage is tracked per request
✅ All API endpoints work as specified
✅ Error handling covers edge cases
✅ Documentation is complete

Environment Setup Commands

```bash
# Create project
mkdir codebase-ai-assistant
cd codebase-ai-assistant

# Create virtual environment
python -m venv venv
source venv/bin/activate  # Mac/Linux
# or venv\Scripts\activate  # Windows

# Install dependencies
pip install -r requirements.txt

# Set up environment variables
cp .env.example .env
# Edit .env with your keys

# Run the application
python app.py
```

Important Notes

1. **Start Simple:** Focus on getting basic repository indexing and question answering working first
2. **Use Haiku 4.5:** It's the cheapest model and sufficient for most tasks
3. **Implement Caching:** This saves 90% on costs - implement from day 1
4. **Track Costs:** Monitor your Anthropic dashboard daily during development
5. **Test with Real Repo:** Use the healthcare insurance API from Phase 1
6. **Document Everything:** This will help when building the frontend later

Next Steps After Completion

Once Phase 2 backend is complete:

1. Test thoroughly with Postman/Thunder Client
2. Document all API endpoints
3. Create example requests/responses
4. Prepare for Phase 3: Frontend development in Lovable
5. Consider adding features like:

- Multi-repository support
- Code review suggestions
- Test generation
- Documentation generation

Remember: This is a POC, so focus on core functionality first. You can always add more features later!