Software Code Bug Detection & Fixing

by

Karthik Reddy Surkanti (23951A6677)
Nikhilesh Kamalapurkar (23951A6675)
Tejal Vinod Kangandul (23951A6676)

Intel Unnati Industrial Training Program 2025

Project Mentor :- Dr Sreelakshmi Doma

GitHub Repository Link:

https://github.com/Kar6677thik/AI_Bug_Detection_Fixing.git

Table of Contents

I. Abstract

The AI Bug Detection System is an offline, AI-powered solution designed to identify and fix bugs in source code while optimized for low-end computing environments. Leveraging the DeepSeek Coder 1.3B model with efficient quantization techniques, the system performs comprehensive code analysis without requiring cloud services or high-performance hardware. The application features a modern web interface that allows developers to submit code for analysis and receive detailed bug reports with confidence-scored fix suggestions. Built on a modular architecture with FastAPI backend and a lightweight frontend, the system currently supports Python and JavaScript analysis with an emphasis on accessibility and performance. Through model quantization, memory offloading, and automatic device mapping, the system achieves the balance of maintaining analytical accuracy while operating within resource constraints. This project demonstrates how AI capabilities can be democratized for code quality improvement across diverse development environments, regardless of hardware limitations.

1. Introduction

The AI Bug Detection System is an offline-capable tool that leverages artificial intelligence to detect and fix bugs in source code. This system is specifically optimized for low-end machines through model quantization techniques, making it accessible to developers with limited computational resources.

1.1 Purpose

The primary purpose of this system is to provide developers with an efficient means of identifying bugs in their code without requiring powerful hardware or cloud services. By using optimized AI models that can run locally, the system offers:

- Automated bug detection in Python and JavaScript
- Suggested fixes for identified issues
- Confidence scores for each suggested fix
- An intuitive web interface for code analysis
- API endpoints for programmatic access

1.2 System Overview

The AI Bug Detection System follows a modular architecture pattern with two main components:

1. **Backend**: Handles the core functionality through:
    - AI Model management (loading, inference)
    - Bug detection and fix suggestion algorithms
    - API server for handling requests
    - Code preprocessing utilities
2. **Frontend**: Provides a user interface for:
    - Code input and submission
    - Results visualization
    - Analysis reports

The system uses the DeepSeek Coder model (1.3B parameters) as its foundation, applying quantization techniques to reduce memory requirements and optimize performance on less powerful machines.

2. System Architecture

The system follows a client-server architecture with a well-defined separation between the frontend and backend components. This section details the architecture, components, and their interactions.

2.1 High-Level Architecture

The AI Bug Detection System is structured as follows:

```
AI_Bug_Detection/
├── backend/
│   ├── api/             # API server and endpoints
│   └── model/           # AI models and inference logic
├── frontend/            # Web interface
└── tests/               # Unit and integration tests
```

2.2 Backend Components

The backend is composed of several key modules:

2.2.1 API Layer (`backend/api/`)

The API layer handles HTTP requests and serves as the interface between the frontend and the AI models. It's built using FastAPI, which provides:

- Fast performance
- Automatic OpenAPI documentation
- Request validation
- Exception handling

Key components:

- `app.py`: Main application entry point with core endpoints
- `routes.py`: Additional API routes for specific functionality

2.2.2 Model Layer (`backend/model/`)

The model layer handles the AI-related functionality:

- `bug_detection.py`: Contains the `BugDetector` class for identifying bugs in code
- `bug_fixer.py`: Contains the `BugFixer` class for suggesting fixes
- `inference.py`: Provides the `InferenceEngine` class that orchestrates detection and fixing
- `preprocess.py`: Contains the `CodePreprocessor` class for preparing code for analysis

2.2.3 Key Classes

BugDetector

- Loads and initializes the DeepSeek Coder model
- Implements quantization for low-end machine optimization
- Provides methods to detect bugs in code

BugFixer

- Extends BugDetector to leverage the same model
- Generates code fixes based on detected bugs
- Provides confidence scores for suggested fixes

InferenceEngine

- Orchestrates the bug detection and fixing process
- Integrates multiple model components

CodePreprocessor

- Handles language-specific code preprocessing
- Removes comments and normalizes code for better analysis

2.3 Frontend Components

The frontend provides a user-friendly interface for interacting with the system:

- `index.html`: Main UI structure and layout
- `main.js`: Handles interface logic and API communication
- `styles.css`: Defines styling for the interface

The frontend features:

- Code input area with line counting
- Analysis result display
- Bug severity indicators
- Export functionality
- Status indicators

2.4 Data Flow

1. User inputs code through the web interface
2. Frontend sends code to the API server
3. API server invokes the InferenceEngine
4. InferenceEngine orchestrates:
    - Code preprocessing
    - Bug detection
    - Fix generation
5. Results are returned to the frontend

6. Frontend displays the results to the user

3. AI Model Architecture and Implementation

This section details the AI models used in the system, their configuration, and implementation specifics.

3.1 Model Selection and Configuration

The system uses the DeepSeek Coder model (1.3B parameters) as its foundation:

```
model_name = "deepseek-ai/deepseek-coder-1.3b-base"
```

This model was chosen for several key reasons:

- Good balance of size and performance
- Specialized for code understanding and generation
- Suitable for quantization and optimization

3.2 Model Optimization for Low-End Machines

Several techniques are employed to optimize the model for low-end machines:

3.2.1 Quantization

The model is loaded with reduced precision using PyTorch's float16 datatype:

```
self.model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    low_cpu_mem_usage=True,
    device_map="auto",
    offload_folder=offload_dir
)
```

This halves the memory requirements compared to standard float32 precision.

3.2.2 Memory Offloading

A dedicated offload folder is created to allow parts of the model to be stored on disk:

```
offload_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)),
"model_offload")
os.makedirs(offload_dir, exist_ok=True)
```

This enables processing on machines with limited RAM by swapping less frequently used model parts to disk.

3.2.3 Device Mapping

The `device_map="auto"` parameter allows the model to automatically distribute its layers across available hardware:

- If a GPU is available, it will use it
- Otherwise, it falls back to CPU

- Can also distribute across multiple devices if available

3.3 Bug Detection Implementation

The bug detection process follows these steps:

1. **Prompt Engineering**: The code is wrapped in a specific prompt template:

   ```
   f"Analyze this code for bugs:\n```\n{code}\n```\nBugs:"
   ```

2. **Inference**: The model generates a response based on the prompt:

   ```
   with torch.no_grad():
       outputs = self.model.generate(
           **inputs,
           max_new_tokens=200,
           temperature=0.7,
           do_sample=True
       )
   ```

3. **Output Parsing**: The raw model output is parsed into a structured bug report:

   ```
   analysis = self.tokenizer.decode(outputs[0], skip_special_tokens=True)
   return self._parse_analysis(analysis)
   ```

3.4 Bug Fixing Implementation

The bug fixing process extends the detection approach:

1. **Fix-Specific Prompt**: For each detected bug, a fix-specific prompt is created:

   ```
   f"Fix this bug in the code:\n```\n{code}\n```\nBug: {bug['description']}\
   nFixed Code:"
   ```

2. **Fix Generation**: The model generates potential fixes:

   ```
   with torch.no_grad():
       outputs = self.model.generate(
           **inputs,
           max_new_tokens=400,
           temperature=0.5,
           do_sample=True
       )
   ```

3. **Confidence Scoring**: A confidence score is calculated for each fix:

   ```
   def _calculate_confidence(self, fix: str) -> float:
       # Simple placeholder - would implement proper scoring
       return 0.8
   ```

3.5 Code Preprocessing

Before analysis, code undergoes preprocessing to improve model performance:

1. **Language-Specific Handlers**: Different preprocessing for different languages:

   ```
   self.language_handlers = {
       'python': self._preprocess_python,
       'javascript': self._preprocess_javascript,
       'default': self._preprocess_generic
   ```

```
}
```

2. **Comment Removal**: Comments are removed to focus on functional code:

   ```
   # Remove comments and docstrings
   cleaned_code = self._remove_comments_and_docstrings(code)
   ```

3. **Whitespace Normalization**: Whitespace is normalized for consistent analysis:

   ```
   # Normalize whitespace
   cleaned_code = '\n'.join(line.strip() for line in cleaned_code.split('\n')
   if line.strip())
   ```

4. API Implementation and Endpoints

This section details the API implementation, available endpoints, request/response formats, and error handling mechanisms.

4.1 API Server Configuration

The API server is built using FastAPI, providing a modern, fast, and scalable web framework:

```
app = FastAPI(
    title="AI Bug Detection API",
    description="API for detecting and fixing bugs in source code",
    version="0.1.0"
)
```

CORS (Cross-Origin Resource Sharing) is configured to allow requests from any origin:

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

4.2 Request Models

A Pydantic model defines the expected request format:

```
class CodeRequest(BaseModel):
    code: str
    language: str = "python"
```

Key fields:

- `code`: The source code to analyze (required)
- `language`: Programming language of the code (default: "python")

4.3 Core Endpoints

4.3.1 `/analyze` Endpoint

Provides full code analysis with bug detection and fix suggestions:

```
@app.post("/analyze")
async def analyze_code(request: CodeRequest):
    """Analyze code for bugs and suggest fixes"""
    if not MODEL_LOADED:
        return {
            "success": False,
            "error": "Model dependencies not installed"
        }
    try:
        analysis = engine.analyze_code(request.code)
        return {
            "success": True,
            "results": analysis
        }
    except Exception as e:
        return {
            "success": False,
            "error": str(e)
        }
```

4.3.2 `/detect` Endpoint

Dedicated endpoint for bug detection only:

```
@router.post("/detect")
async def detect_bugs(request: CodeRequest):
    """Endpoint specifically for bug detection"""
    try:
        bug_report = engine.detector.detect_bugs(request.code)
        return {
            "success": True,
            "bugs": bug_report.get("bugs", []),
            "warnings": bug_report.get("warnings", []),
            "analysis": bug_report.get("analysis", "")
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

4.3.3 `/fix` Endpoint

Dedicated endpoint for generating fixes:

```
@router.post("/fix")
async def fix_bugs(request: CodeRequest):
    """Endpoint specifically for generating fixes"""
    try:
        bug_report = engine.detector.detect_bugs(request.code)
        fixes = engine.fixer.suggest_fixes(request.code, bug_report)
        return {
            "success": True,
            "fixes": fixes
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

4.3.4 `/full-analysis` Endpoint

Comprehensive endpoint combining detection and fixing:

```
@router.post("/full-analysis")
async def full_analysis(request: CodeRequest):
    """Endpoint combining detection and fixing"""
    try:
        results = engine.analyze_code(request.code)
        return {
            "success": True,
            "results": results
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

4.3.5 Utility Endpoints

- `/health`: Health check endpoint
- `/test`: Test endpoint to verify server operation

4.4 Response Format

Responses follow a consistent format:

```
{
  "success": true,
  "results": {
    "bug_report": {
      "bugs": [
        {
          "description": "Bug description",
          "line": 5,
          "severity": "critical"
        }
      ],
      "warnings": [],
      "analysis": "Full analysis text"
    },
    "fixes": [
      {
        "bug": {
          "description": "Bug description",
          "line": 5
        },
        "fix": "Suggested fix code",
        "confidence": 0.85
      }
    ]
  }
}
```

4.5 Error Handling

The API implements comprehensive error handling:

1. **Try-Except Blocks**: All endpoint handlers use try-except blocks to catch errors:

   ```
   try:
       analysis = engine.analyze_code(request.code)
       # Handle success
   ```

```
except Exception as e:
    # Handle error
```

2. **HTTP Exception Handling**: FastAPI's `HTTPException` class is used for proper HTTP error responses:

   ```
   raise HTTPException(status_code=500, detail=str(e))
   ```

3. **Graceful Fallbacks**: When model dependencies aren't available, the system provides clear error messages:

   ```
   if not MODEL_LOADED:
       return {
           "success": False,
           "error": "Model dependencies not installed"
       }
   ```

5. Frontend Implementation and User Interface

This section details the frontend implementation, user interface design, and client-side functionality.

5.1 User Interface Design

The frontend provides a modern, intuitive interface styled with Tailwind CSS and Font Awesome icons. The main components include:

5.1.1 Header Section

```
<div class="bg-gray-700 p-5 border-b border-gray-600">
  <div class="flex items-center justify-between">
    <div class="flex items-center">
      <i class="fas fa-bug text-blue-400 text-2xl mr-3"></i>
      <h1 class="text-3xl font-bold text-blue-400">AI Bug Detection System</h1>
    </div>
    <div class="flex space-x-2">
      <div class="w-3 h-3 rounded-full bg-red-500"></div>
      <div class="w-3 h-3 rounded-full bg-yellow-500"></div>
      <div class="w-3 h-3 rounded-full bg-green-500"></div>
    </div>
  </div>
</div>
```

The header features:

- System logo and title
- Decorative indicators resembling macOS window controls for visual appeal

5.1.2 Code Input Area

```
<div class="mb-6">
  <div class="flex justify-between items-center mb-2">
    <div class="flex items-center">
      <i class="fas fa-code text-gray-400 mr-2"></i>
```

```
      <label for="codeInput" class="text-gray-300 font-semibold">Source
Code</label>
    </div>
    <div class="text-xs text-gray-400 bg-gray-700 px-2 py-1 rounded">
      <span id="lineCount">0</span> lines
    </div>
  </div>
  <div class="relative">
    <textarea id="codeInput"
      class="w-full bg-gray-900 text-green-400 font-mono rounded-lg resize-none
p-4 border border-gray-700 focus:border-blue-500 focus:ring-1 focus:ring-blue-
500 focus:outline-none h-48"
      placeholder="Paste your code here..."></textarea>
    <button id="clearButton" class="absolute top-2 right-2 text-gray-500
hover:text-gray-300">
      <i class="fas fa-times-circle"></i>
    </button>
  </div>
</div>
```

The code input area includes:

- Monospaced textarea for code entry
- Line counter display
- Clear button for quick text removal
- Syntax highlighting design with green text on dark background

5.1.3 Control Buttons

```
<div class="flex items-center justify-between mb-6">
  <div class="flex space-x-3">
    <button id="analyzeButton"
      class="bg-blue-600 hover:bg-blue-700 text-white font-medium px-6 py-2
rounded-lg transition duration-300 flex items-center">
      <i class="fas fa-search-plus mr-2"></i>
      Analyze Code
    </button>
    <button id="downloadButton"
      class="bg-gray-700 hover:bg-gray-600 text-gray-300 font-medium px-4 py-2
rounded-lg transition duration-300 flex items-center">
      <i class="fas fa-download mr-2"></i>
      Export Results
    </button>
  </div>
  <div class="hidden sm:flex items-center space-x-4">
    <!-- Severity indicators -->
  </div>
</div>
```

Control buttons include:

- "Analyze Code" button to initiate analysis
- "Export Results" button for saving analysis results
- Visual severity indicators for critical, warning, and info-level issues

5.1.4 Results Area

```
<div>
  <div class="flex justify-between items-center mb-2">
    <div class="flex items-center">
      <i class="fas fa-clipboard-list text-gray-400 mr-2"></i>
      <h3 class="text-gray-300 font-semibold">Analysis Results</h3>
    </div>
    <div id="statusBadge" class="bg-gray-700 text-gray-300 text-xs px-2 py-1 rounded-full">
      Ready
    </div>
  </div>
  <div id="results"
    class="w-full bg-gray-900 rounded-lg p-4 border border-gray-700 min-h-[120px] max-h-[300px] overflow-auto break-words whitespace-pre-wrap font-mono text-sm">
    <div class="flex items-center text-gray-400">
      <i class="fas fa-info-circle mr-2"></i>
      <p>Analysis results will appear here...</p>
    </div>
  </div>
</div>
```

The results area features:

- Status badge showing current operation state
- Scrollable container for analysis results
- Monospaced text formatting for code snippets
- Default placeholder message

5.2 Client-Side Functionality

The `main.js` file handles client-side functionality:

5.2.1 Event Listeners

```
document.addEventListener('DOMContentLoaded', function() {
    const codeInput = document.getElementById('codeInput');
    const analyzeButton = document.getElementById('analyzeButton');
    const resultsDiv = document.getElementById('results');

    analyzeButton.addEventListener('click', async function() {
        // Code analysis event handler
    });

    // Other event listeners
});
```

Event listeners manage:

- DOM content loading initialization
- Analyze button clicks
- Code input changes
- Clear button functionality

5.2.2 API Communication

```
try {
    const response = await fetch('https://e0ee-35-227-72-66.ngrok-free.app/full-analysis', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({
            code: code,
            language: 'python' // Default to Python for now
        })
    });

    const data = await response.json();
    displayResults(data);
} catch (error) {
    resultsDiv.innerHTML = `<div class="error">Error: ${error.message}</div>`;
}
```

API communication includes:

- Asynchronous fetch requests
- JSON request formatting
- Error handling with user-friendly messages
- Currently accesses an Ngrok endpoint (likely for development)

5.2.3 Results Display

```
function displayResults(data) {
    if (!data.success) {
        resultsDiv.innerHTML = `<div class="error">${data.error || 'Unknown
error occurred'}</div>`;
        return;
    }

    let html = '';
    const results = data.results;

    // Generate HTML for bugs
    if (results.bug_report.bugs && results.bug_report.bugs.length > 0) {
        html += '<h2>Detected Bugs</h2>';
        // Bug display logic
    } else {
        html += '<h2>No bugs detected!</h2>';
        // Analysis display logic
    }

    // Generate HTML for fixes
    if (results.fixes && results.fixes.length > 0) {
        html += '<h2>Suggested Fixes</h2>';
        // Fix display logic
    }

    resultsDiv.innerHTML = html;
}
```

The results display:

- Formats bug information in cards
- Shows fix suggestions with confidence scores
- Gracefully handles the absence of bugs
- Displays raw analysis when no structured bugs are found

6. Installation and Deployment Guide

6.1 System Requirements

The AI Bug Detection System is designed to run on low-end machines with the following minimum specifications:

- CPU: Dual-core processor, 2.0 GHz or higher
- RAM: 4 GB minimum (8 GB recommended)
- Storage: 2 GB of free disk space
- Operating System: Windows 10/11, macOS 10.15+, or Linux (Ubuntu 20.04+)
- Python: Version 3.8 or higher

6.2 Installation Steps

6.2.1 Clone the Repository

```
git clone https://github.com/Kar6677thik/AI_Bug_Detection_Fixing.git
cd AI_Bug_Detection
```

6.2.2 Virtual Environment Setup

Creating and activating a virtual environment is recommended to avoid package conflicts:

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

# Activate virtual environment
# For Linux/Mac
source venv/bin/activate
# For Windows
venv\Scripts\activate
```

6.2.3 Installing Dependencies

Install all required packages using pip:

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

This will install the necessary packages including FastAPI, Uvicorn, PyTorch, Transformers, and other dependencies.

6.3 Configuration

The system can be configured through several key files:

6.3.1 Model Configuration

Edit `backend/model/bug_detection.py` to:

- Change the model being used (default is "deepseek-ai/deepseek-coder-1.3b-base")
- Adjust quantization parameters
- Modify inference settings like temperature and token limits

Example configuration change:

```
# Change model or adjust parameters
```

```
self.model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,  # Change precision if needed
    low_cpu_mem_usage=True,
    device_map="auto",
    offload_folder=offload_dir
)
```

6.3.2 API Configuration

In `backend/api/app.py`, you can configure:

- CORS settings for security
- API base path
- Version information

6.4 Running the System

6.4.1 Starting the Backend Server

To start the FastAPI server:

```
# From the project root directory
uvicorn backend.api.app:app --reload --host 0.0.0.0 --port 8000
```

The `--reload` flag enables auto-reloading during development, and can be removed in production.

6.4.2 Accessing the Web Interface

The frontend can be accessed in two ways:

1. **Direct file access**: Open `frontend/index.html` in a web browser
2. **Using a local server**: Set up a simple HTTP server:

   ```
   # From the frontend directorypython -m http.server 8080
   ```

   Then access the interface at http://localhost:8080

6.4.3 Production Deployment

For production deployment, consider:

- Using Gunicorn with Uvicorn workers for better performance
- Setting up Nginx as a reverse proxy
- Implementing proper security measures for the API
- Hosting the frontend files on a proper web server

Example production server start command:

```
gunicorn backend.api.app:app -w 4 -k uvicorn.workers.UvicornWorker --bind
0.0.0.0:8000
```

7. Testing Framework

7.1 Testing Structure

The project implements a comprehensive testing strategy with tests located in the `tests/` directory. The testing framework uses Python's built-in `unittest` module.

7.2 API Tests

API tests (`test_api.py`) verify the correct functioning of all API endpoints:

- Health check endpoint
- Analysis endpoint
- Detection endpoint
- Fix endpoint

These tests ensure that:

- Endpoints return the expected status codes
- Response data structures are correct
- Error handling works as expected

7.3 Model Tests

Model tests (`test_model.py`) verify the functionality of:

- Bug detection capabilities
- Fix suggestion generation
- Model loading and inference

These tests ensure that the model components can:

- Correctly identify common bugs in code
- Generate appropriate fix suggestions
- Handle edge cases and unexpected inputs

7.4 Running Tests

To run all tests:

```
python -m unittest discover tests
```

To run specific test files:

```
python -m unittest tests.test_api
python -m unittest tests.test_model
```

8. Performance Optimizations

8.1 Model Optimizations

The system employs several techniques to optimize model performance on low-end hardware:

8.1.1 Model Quantization

Float16 precision is used instead of float32, reducing memory usage by approximately 50% with minimal impact on accuracy.

8.1.2 Memory Offloading

The system implements disk offloading for model weights that aren't actively used, reducing peak memory usage.

8.1.3 Automatic Device Mapping

The `device_map="auto"` setting allows the system to intelligently distribute model components between available hardware (CPU/GPU).

8.2 API Optimizations

- Asynchronous request handling
- Proper error handling to prevent crashes
- Response caching for common requests

8.3 Frontend Optimizations

- Minimal JavaScript dependencies
- Efficient DOM manipulation
- Responsive design for various screen sizes

9. Known Limitations and Future Work

9.1 Current Limitations

- Limited support for programming languages beyond Python and JavaScript
- Fixed model size (1.3B parameters) as a compromise between accuracy and performance
- Basic code analysis that may miss complex bugs involving multiple files
- Limited support for framework-specific bugs (e.g., React, Django)

9.2 Future Enhancements

9.2.1 Technical Improvements

- Add support for more programming languages
- Implement more sophisticated bug classification
- Add static analysis tools integration
- Implement user feedback loop for improving fix suggestions

9.2.2 Usability Improvements

- Add user accounts and history tracking
- Implement project-wide code analysis
- Add customizable detection rules
- Create desktop application with Electron

10. Conclusion

The AI Bug Detection System demonstrates how AI can be leveraged to improve code quality and developer productivity, even on resource-constrained hardware. By combining transformer-based language models with optimized inference techniques, the system provides valuable bug detection and fixing capabilities without requiring cloud services or high-end hardware.

The modular architecture allows for easy extensibility and customization, while the optimization techniques ensure reasonable performance even on lower-end machines. As the field of AI for code continues to advance, this system provides a solid foundation that can be enhanced with improved models and additional features.