🚀 DevDost Self-Healing AI Agent System

Overview

This is a **production-grade, autonomous full-stack AI agent** that automatically creates, debugs, and runs web projects without any manual intervention. The agent **never stops on errors** and includes intelligent auto-recovery mechanisms.

🎯 Key Features

1. Self-Healing Architecture

- ✅ Automatic error detection and recovery
- ✅ Up to 3 retry attempts with intelligent debugging
- ✅ No manual intervention required
- ✅ Treats errors as tasks to solve, not reasons to stop

2. Zero KeyError Policy

- ✅ `normalize_state()` ensures all dictionary keys exist
- ✅ Automatic fallback values for missing state
- ✅ No NoneType or KeyError crashes

3. Auto Tech Stack Detection

- ✅ AI decides React vs Next.js vs Node vs HTML/CSS
- ✅ No user input required for tech choice
- ✅ Automatically infers project structure

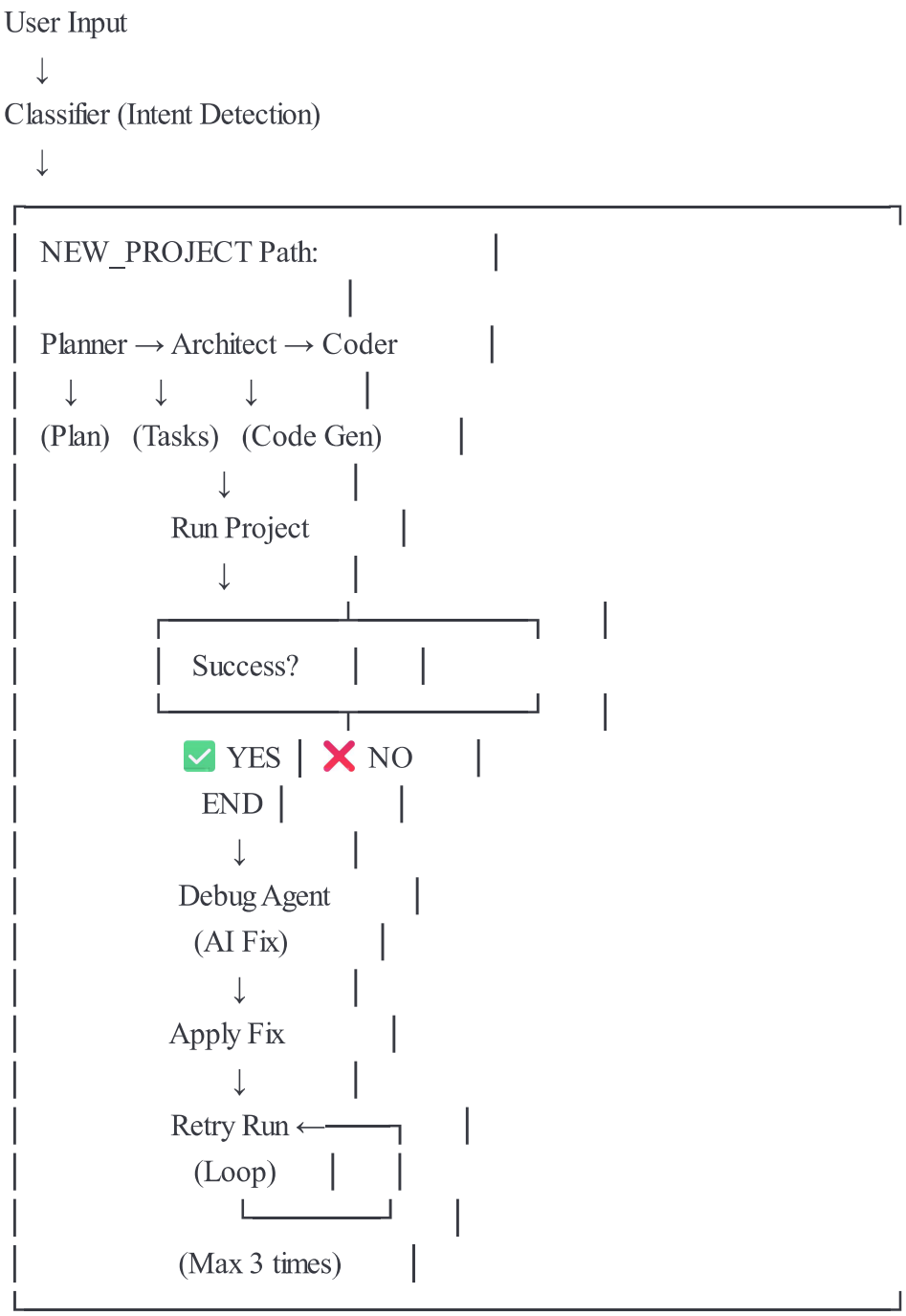
4. Real-Time Progress Streaming

- ✅ Step-by-step progress messages
- ✅ WebSocket-based live updates
- ✅ Concise status messages (not paragraphs)

5. Automatic Run + Debug Loop

- ✅ Auto-runs project after code generation
- ✅ Captures runtime errors and logs
- ✅ AI analyzes errors → generates fixes → retries
- ✅ Continues until success or max retries

📊 Agent Flow



🧩 Component Breakdown

graph.py - Core Agent System

Key Agents:

1. `chat_classifier_agent`
    - Classifies user intent (NEW_PROJECT, MODIFY_PROJECT, etc.)
    - Routes to appropriate handler
2. `planner_agent`
    - Generates complete project plan
    - Auto-detects best tech stack
    - Creates file structure plan
3. `architect_agent`
    - Breaks plan into implementation tasks
    - Orders tasks by dependency
    - Provides detailed task instructions
4. `coder_agent`
    - Implements each task sequentially
    - Writes COMPLETE code (no placeholders)
    - Tracks progress with status updates
5. `run_project_agent` ⭐ NEW
    - Attempts to run the project
    - Captures stdout/stderr logs
    - Detects runtime errors
    - Returns `SUCCESS` or `RUN_FAILED`
6. `debug_agent` ⭐ NEW
    - AI-powered error analysis
    - Generates targeted fixes:
        - File updates (syntax, imports)
        - Package installation
        - File creation/deletion
    - Tracks debug history
7. `general_chat_agent`
    - Handles casual conversation
    - Provides help and guidance

app.py - Backend Server

Key Features:

- **Real-time progress via WebSocket**
    - `emit_progress()` sends updates to frontend
    - Events: `ai_progress`, `agent_complete`, `project_status`
- **Async agent execution**
    - Agent runs in background thread
    - Frontend receives immediate acknowledgment
    - Progress streams during execution
- **Project management**
    - Create, list, switch, delete projects
    - File operations (CRUD)
    - Download projects as ZIP

prompts.py - AI Prompts

Enhanced prompts for:

- Planning with auto tech detection
- Architecture with dependency ordering
- Coding with completeness requirements
- Debugging with error analysis

states.py - State Models

All state models with:

- Type safety via Pydantic
- Default values to prevent KeyErrors
- Clear field descriptions

🔄 Self-Healing Mechanism

Error Detection

python

```python
def run_project_agent(state: dict) -> dict:
    try:
        # Run project based on type
        if project_structure == "react":
            result = subprocess.Popen(["npm", "start"], ...)

        if result.poll() is None:  # Success
            return {**state, "status": "SUCCESS"}
        else:
            raise Exception(f"Run failed: {stderr}")
    except Exception as e:
        return {**state, "status": "RUN_FAILED", "last_error": str(e)}
```

Automatic Debug + Fix

python

```python
def debug_agent(state: dict) -> dict:
    last_error = state.get("last_error")
    run_attempts = state.get("run_attempts", 0)

    if run_attempts >= MAX_RETRIES:
        return {**state, "status": "DONE"}  # Give up after 3 tries

    # AI analyzes error
    fix_plan = llm.invoke(debug_prompt(last_error, files, history))

    # Apply fix based on type
    if fix_type == "file_update":
        create_file_tool.run({...})  # Update file
    elif fix_type == "install_package":
        subprocess.run(["npm", "install", package])

    return {
        **state,
        "status": "RETRY_RUN",
        "run_attempts": run_attempts + 1
    }
```

Retry Loop

python

```python
# In graph routing
def route_after_run(state: dict) -> str:
    if status == "SUCCESS":
        return "END"  # Done!
    if status == "RUN_FAILED":
        return "debug"  # Go fix it

def route_after_debug(state: dict) -> str:
    return "run_project"  # Try again
```

💡 Usage Examples

Example 1: Simple Request

User: "create a todo app"

AI Flow:
1. Classifier: NEW_PROJECT
2. Planner: Decides React, creates plan
3. Architect: 8 implementation tasks
4. Coder: Creates App.js, Todo.js, styles...
5. Run: npm start
6. Success: "✅ Running at http://localhost:3000"

Example 2: Error Recovery

User: "build a landing page"

AI Flow:
1-4. [Creates Next.js project]
5. Run: npm run dev
   ❌ Error: "Module 'next' not found"
6. Debug: Diagnosis = "Missing dependencies"
   Fix = Run "npm install"
7. Run: npm run dev (retry 1)
   ✅ Success: "Running at http://localhost:3000"

Example 3: Syntax Error Fix

User: "make a calculator"

AI Flow:
1-4. [Creates HTML/CSS/JS project]
5. Run: python -m http.server
❌ Error: "SyntaxError: missing) in script.js line 42"
6. Debug: AI finds syntax error
Fix = Updates script.js with correct syntax
7. Run: Retry
✅ Success: "Running at http://localhost:8000"

🛠️ Configuration

Environment Variables

bash

```
GROQ_API_KEY=your_groq_api_key
```

Adjustable Settings (graph.py)

python

```
MAX_RETRIES = 3  # Max debug attempts
```

Supported Project Types

- ✅ HTML/CSS/JavaScript (static sites)
- ✅ React (create-react-app)
- ✅ Next.js (with TypeScript + Tailwind)
- ✅ Node.js/Express (backend APIs)
- ✅ Python/Flask/Django

📡 WebSocket Events

From Backend to Frontend:

- `ai_progress` - Step-by-step progress

json


```
{
  "message": "⚙️ Creating App.js (3/8)",
  "project": "todo-app",
  "timestamp": 1234567890
}
```


- `agent_complete` - Final result

json


```
{
  "success": true,
  "message": "✅ Running at http://localhost:3000",
  "current_project": "todo-app",
  "status": "SUCCESS"
}
```


- `project_status` - Run status updates

json


```
{
  "project": "todo-app",
  "status": "running",
  "url": "http://localhost:3000"
}
```

🎨 Frontend Integration

javascript

```javascript
// Connect to WebSocket
const socket = io('http://localhost:5000');

// Listen for progress
socket.on('ai_progress', (data) => {
  console.log(data.message);
  // Update UI with progress
});

// Listen for completion
socket.on('agent_complete', (result) => {
  console.log('Done!', result.message);
  // Show final result
});

// Send chat message
fetch('http://localhost:5000/chat', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    message: 'create a portfolio website',
    session_id: 'user123'
  })
});
```

🚨 Error Handling Philosophy

Traditional Approach (WRONG ❌)

python

```
try:
    result = run_project()
    if result.error:
        return "Error occurred. Please fix manually."
```

Self-Healing Approach (CORRECT ✅)

python

```
for attempt in range(MAX_RETRIES):
    try:
        result = run_project()
        if result.success:
            return "Success!"
        else:
            error = result.error
            fix = ai_debug(error)
            apply_fix(fix)
            continue  # Try again
    except Exception as e:
        # Even exceptions trigger auto-fix
        fix = ai_debug(str(e))
        apply_fix(fix)
        continue
```

🎯 Production Best Practices

1. **State Normalization**
   - Always call `normalize_state()` first
   - Provides default values for all keys
2. **Progress Messages**
   - Keep concise (5-10 words)
   - Use emojis for clarity
   - Example: "⚙️ Creating App.js (3/8)"
3. **Error Messages**
   - Truncate long errors ([:200])
   - Focus on actionable info
4. **Timeouts**
   - Set reasonable timeouts for subprocess calls
   - npm install: 180s

    - npm init: 30s
    - Run checks: 2-3s
5. **Resource Cleanup**
    - Always stop processes on exit
    - Use `atexit.register(cleanup)`

📈 Performance Notes

- **Average project creation**: 30-60 seconds
- **React/Next.js**: 2-3 minutes (includes npm install)
- **HTML projects**: 5-10 seconds
- **Debug cycles**: 10-20 seconds per attempt
- **Max total time**: ~5 minutes (3 retries)

🔮 Future Enhancements

- [ ] Support for Vue.js, Svelte
- [ ] Database setup (MongoDB, PostgreSQL)
- [ ] Deployment automation (Vercel, Netlify)
- [ ] Git integration
- [ ] Collaborative editing
- [ ] AI code review before run
- [ ] Performance optimization suggestions

🤝 Contributing

This system is designed to be:

- **Extensible**: Add new project types easily
- **Maintainable**: Clear separation of concerns
- **Testable**: Each agent is independent
- **Observable**: Comprehensive logging

📝 License

MIT License - Feel free to use and modify!

Built with ❤️ by DevDost Team