From Exceptions to Pull Requests: Building a Self Fixing .NET 10 App with AI Agents

A step-by-step guide to automating error analysis and remediation using Azure Monitor, Semantic Kernel, and LLMs.

What if your application could fix itself?

It sounds like SF, but in 2026, it's not. Today, we can build systems that don't just log errors but actively repair them. Imagine a system that:

- Monitors Application Insights 24/7.
- Reads your actual source code to understand context.
- Diagnoses root causes with LLM reasoning.
- Creates Pull Requests automatically with the fix.

I built this exact system. While I used Azure services and .NET 10, the architectural patterns apply to any stack.

In this deep dive, we'll cover:

- The Infrastructure to manage a self-fixing production application
- How it works
- The cost

Try it yourself:

You can find the full source code and Bicep templates here: https://github.com/jBeyondstars/ai-diagnostics-agent

What Makes an AI Agent Different?

Before we dive into code, let's clarify an important distinction: an AI agent is not a chatbot.

A chatbot answers questions. An agent takes actions.

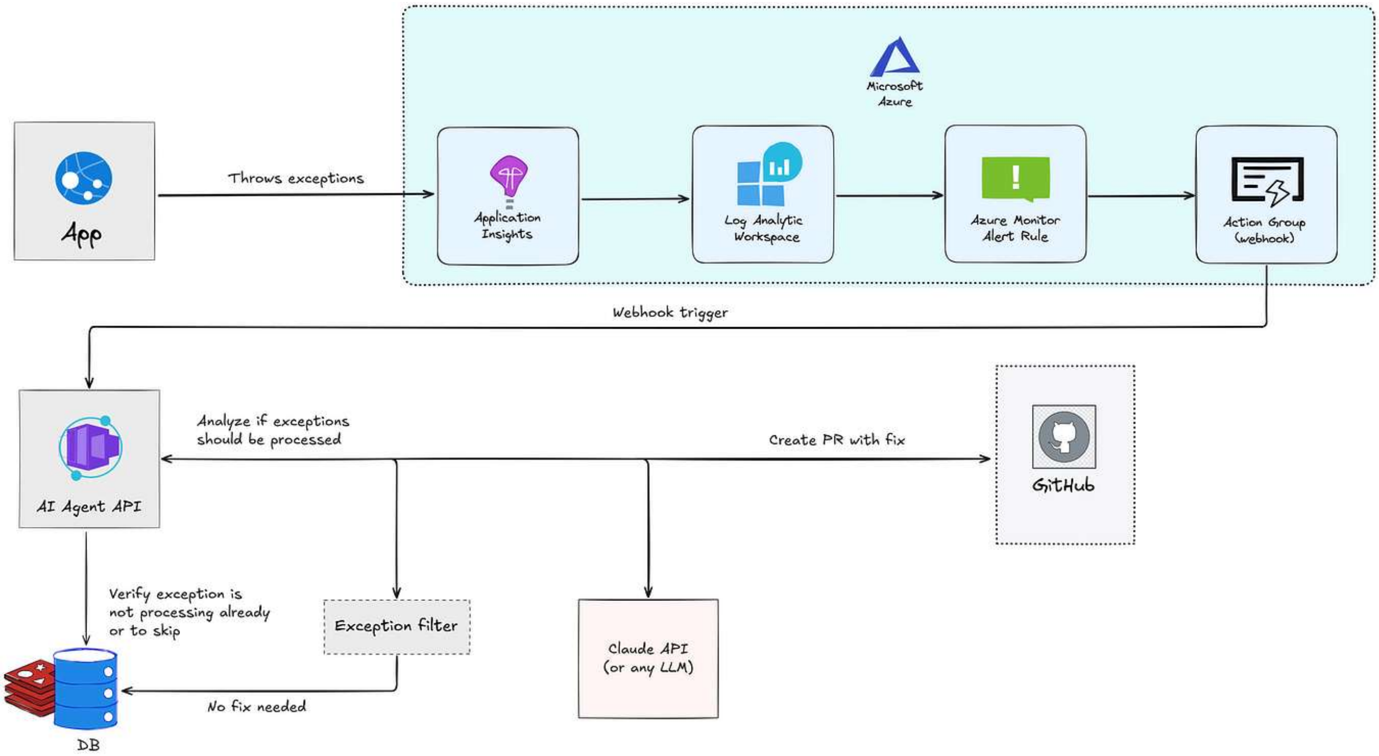
- **Chatbot:** "Here is how you might fix that `NullReferenceException` ..."
- **Agent:** Reads the code, identifies the null object, creates a branch, commits the fix, and opens a Pull Request.

The agent follows a loop: Perceive → Reason → Act → Observe.

It “perceives” the world through tools. It can “see” errors using an `AppInsights_get_exceptions` tool or "read" code using `GitHub_get_file_content`.

We use **Microsoft Semantic Kernel** to bridge the gap between the LLM and these tools.

The Architecture



Simplified Architecture Diagram

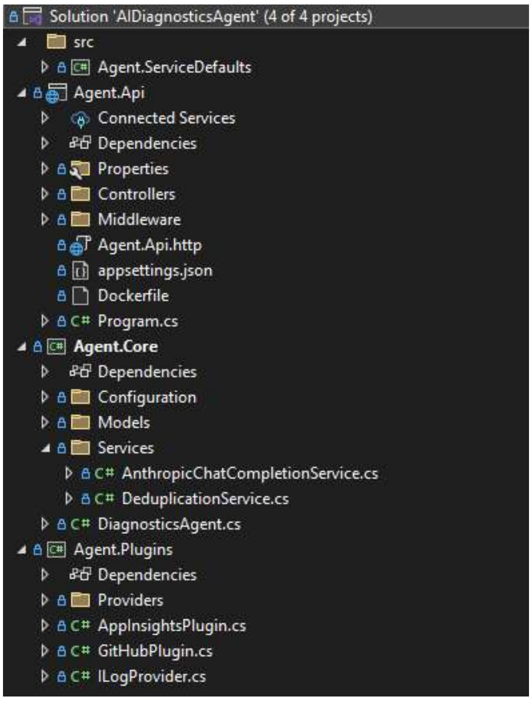
The flow is event-driven:

1. **The App** throws an exception.
2. **Azure Monitor** detects it and triggers an Alert Rule.
3. **The Agent API** receives a webhook, analyzes the issue, and consults the LLM (Claude Opus 4.5/Sonnet 4.5).
4. **GitHub** receives a new Pull Request if a fix is found.

As I said, for this project I used an Azure environment but you could apply the same process in your own setup.
Let’s look at the tech stack:

- .NET 10 App (the one being monitored)
- Azure Monitor & Log Analytics
- .NET 10 Agent App
- Redis for deduplication
- Semantic Kernel to orchestrate the AI Plugins
- Claude API
- A GitHub Repo and Personal Access Token with permissions to allow our agent to create the PRs.

For the demo, I used a single solution for both the App and the agent. It is structured as below:



Agent.Api

The REST API layer that exposes our diagnostic endpoints.

When Azure Monitor detects exceptions, it calls the /analyze webhook endpoint here.

Agent.Core

The brain of the operation. This is where the orchestration happens, among other things:

- Creates and manages the Semantic Kernel agent
- Prevents analyzing the same exception twice

Agent.Plugins

The custom tools that Claude can invoke. I set up a list of tools for communicating with Azure AppInsights and GitHub. Find the list below.

AppInsights Plugin:

Function Name	Description
`get_exceptions`	Retrieves exceptions grouped by type with occurrence counts
`get_exception_details`	Gets detailed samples with full stack traces

GitHub Plugin:

Tool	Purpose
`get_file_content`	Retrieves source file content from the repository
`search_code`	Searches for code snippets. Critical for finding where a class is defined when the stack trace doesn't give the full path.

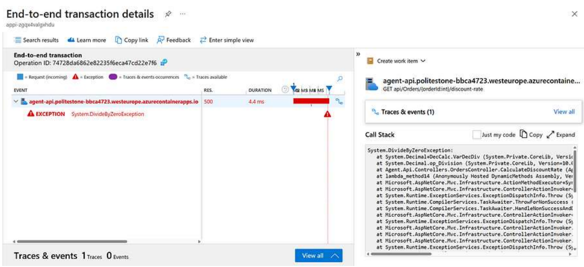
These plugins are registered with Semantic Kernel, allowing the LLM to understand when and how to use them based on the context.

The Flow

Let's look at a concrete example. "Mr. Doe" is having a rough day on our e-commerce site.

He applies a coupon to an empty cart. The app divides by zero items and he gets an error message.

Moments later, Application Insights logs this error.



App Insights interface on exception details (DivideByZeroException).

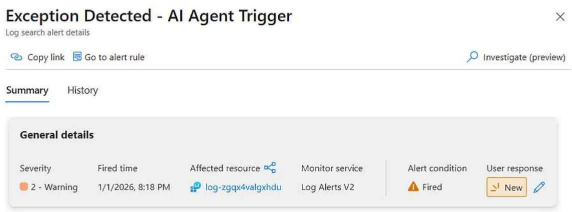
Step 1: The Trigger (Azure Monitor)

We don't want to poll manually. We set up an Azure Monitor Alert Rule running a KQL (Kusto Query Language) query every 5 minutes:

```
AppExceptions
| where TimeGenerated > ago(5m)
| summarize Count = count() by ExceptionType, Pr
| where Count >= 1
```

When this query returns results, it fires a webhook to our Agent's `/analyze` endpoint.

We can verify on the Azure Portal in our Alert Rule history:



Step 2: The Agent (Agent.Core)

The agent receives the alert. Before asking the LLM to fix it, it performs a smart filter:

1. **Deduplication:** Has this exception been processed recently? (Checked via Redis).
2. **Exclusion:** Is it a `TimeoutException` or `SQLConnectionError`? (Infrastructure issues shouldn't trigger code fixes).

If the exception is actionable, we construct a Hybrid Prompt.

> **Note:** I found that pre-fetching the stack trace and relevant code snippets *before* calling the LLM reduces latency and cost significantly, compared to letting the LLM "ask" for every file one by one.

Here is the prompt we are using:

```
"You are an expert .NET developer. Analyze this

{{contextBuilder}}

## Instructions

I have PRE-FETCHED the source code above. Analy

**IMPORTANT**: Only use tools if absolutely nec
- Use `GitHub_get_file_content` if you need to
- Use `GitHub_search_code` if you need to find
- Use `AppInsights_get_exception_details` if yo
- Do NOT call tools if the pre-fetched code is

## Analysis Steps

1. First, analyze the pre-fetched code to ident
2. If you need more context, use the tools
3. Determine if this is an INPUT VALIDATION err
4. If it's a bug, propose a fix

## Response Format

Respond with JSON:
```json
{
    "action": "fix" or "no_fix_needed",
    "rootCause": "explanation",
    "severity": "High|Medium|Low",
    "toolsUsed": ["list of tools you called, or
    "fix": {
        "filePath": "path/to/file.cs",
        "originalCode": "EXACT code to replace"
        "fixedCode": "corrected code",
        "explanation": "what was wrong and how
        "confidence": "High|Medium|Low"
    }
}
```
```

```
    If action is "no_fix_needed", omit the "fix" ob
```

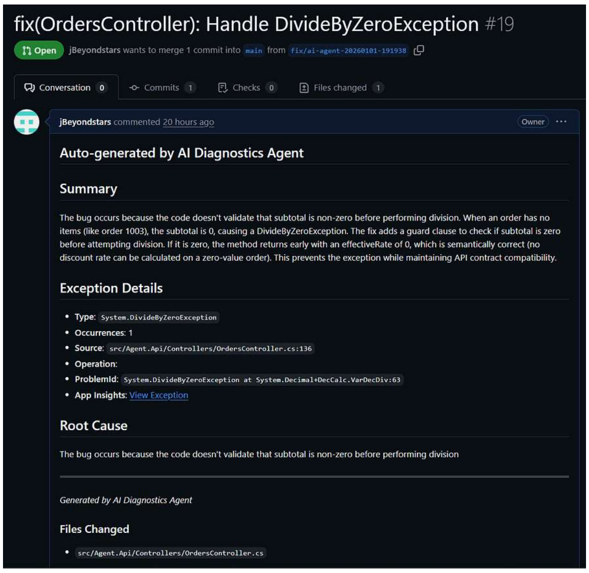
The variable `contextBuilder` contains the details from the exception.

Here we explain to Claude what he tools he can use and how we want him to proceed with the exceptions.
We'll deep dive in the Tools section to understand how Claude recognize how to use the tools.

Step 3: The Fix (Pull Request)

Once the LLM returns the corrected code, the agent creates a branch and opens a PR.

We can detail the PR description and title as much as we want. Our agent currently provide a brief summary, some exception details with a link to the exception on App Insights.



Fully detailed description with App Insights link of the exception

You may wonder know, is the fix viable? Well, if you ever worked with Claude (especially Opus 4.5) you already know how reliable it is in code generation.
So yes, every fixes suggested so far in my tests were consistent and pertinent.

The changes suggested by the Agent for a piece of code trying to call .First() on a collection without verifying in contains elements.

Exception Filtering

As you know, all exceptions don't require a remediation.
HttpRequestException when an external API is down? No code fix.
TimeoutException on a slow database? Infrastructure problem.

The agent uses a 3-level filtering system.

Level 1: Exclusion List Infrastructure and transient exceptions are automatically skipped

```
public string[] ExcludedTypes { get; set; } =
[
"System.Net.Http.HttpRequestException",
"System.TimeoutException",
"System.Threading.Tasks.TaskCanceledException",
"StackExchange.Redis.RedisConnectionException",
"Azure.RequestFailedException",
"Microsoft.Data.SqlClient.SqlException",
    // ...
];
```

Level 2: Pattern Matching Partial string matches catch variants

```
public string[] ExcludedPatterns { get; set; } =
[
"ConnectionException",
"Timeout",
"Unreachable",
"NetworkError",
"HostNotFound"
  // ...
];
```

Level 3: Claude Evaluation

For ambiguous exceptions (containing “api”, “service”, “50*”, …), Claude evaluates.

```
private async Task<bool> EvaluateIfFixableAsync(
{
var prompt =
"You are evaluating if a production exception ca

Exception Type: {exception.ExceptionType}
Message: {exception.Message}
Stack Trace (first 500 chars): {exception.StackT
Operation: {exception.OperationName}

Question: Can this exception be prevented or han

Consider:
- If this is a transient infrastructure error (n
- If the code is missing proper error handling,
- If the exception reveals a bug in the business
- If the exception is caused by external systems

Respond with ONLY one word: YES or NO";
// ..
}
```

…

How Tool Use Works with Claude API

When building an AI agent that can interact with external systems, you might think that simply telling Claude about available tools in the system prompt is enough. It's not.

Here's what actually happens behind the scenes.

1. Tool Definitions

When calling the Claude API, we must pass a tools array containing JSON schemas for each tool. This tells Claude what tools exist and how to call them:

```
{
  "name": "AppInsights_get_exceptions",
  "description": "Retrieves exceptions from th
  "input_schema": {
    "type": "object",
    "properties": {
      "hours": { "type": "integer", "default":
    }
  }
}
```

2. Tool Execution

Our backend must contain the actual code that performs the work. In our case, we use Semantic Kernel plugins:

```
[KernelFunction("get_exceptions")]
  [Description("Retrieves exceptions from the la
  public async Task<string> GetExceptionsAsync(i
  {
      // Actually query Application Insights her
      // Code available in the Github repo
  }
```

What is Kernel?

Semantic Kernel is Microsoft's SDK that acts as a bridge between your code and LLMs. Think of it as the middleware that connects Claude (or any LLM) to your application's capabilities.

In this project we use the Kernel for three purposes:

- LLM Abstraction: lets us swap between Claude, OpenAI, or Azure OpenAI with a simple config change.
- Plugin Registry
- Agent Orchestration: The Kernel is passed to the Agent, which uses it to discover and execute tools.

```
var agent = new ChatCompletionAgent
 {
   Name = "DiagnosticsAgent",
   Instructions = SystemPrompt,
   Kernel = kernel, // Contains all registered p
   Arguments = new KernelArguments(new OpenAIPro
```

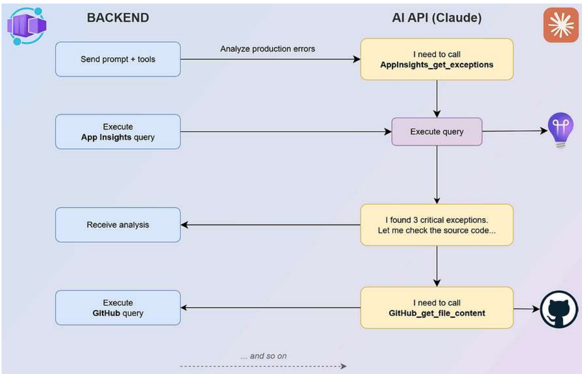
```
        {
        FunctionChoiceBehavior = FunctionChoiceBehavi
        })
    };
```

The FunctionChoiceBehavior.Auto() is the key here. The LLM autonomously decides which tools to invoke based on the conversation context.

3. The Tool Loop

When Claude decides to use a tool, it doesn't execute anything. It simply requests a tool call. Our code must:

1. Detect that Claude wants to use a tool (StopReason = "tool_use")
2. Parse the tool name and arguments
3. Execute the corresponding function
4. Send the result back to Claude
5. Let Claude continue reasoning with the new data



Agent flow diagram

Deduplication Handling

Without deduplication, the same NullReferenceException thrown 50 times would trigger 50 Claude API calls and 50 identical PRs.

Not ideal, right?

We'll use two layers of protection.

1. GitHub PR Check
2. History storage in DB.

No open PR doesn't mean we should re-analyze immediately. The exception might reoccur while the fix is being deployed, or the developer is still reviewing. We enforce a cooldown before allowing re-analysis of the same problem.

Redis vs Database

I used Redis for this demo (simple and fast to implement). For production, I'd recommend adding a database to store fix history: which exceptions were fixed, which PRs were merged vs rejected. This data enables a learning loop for improving the agent over time.

Staying in Sync with GitHub

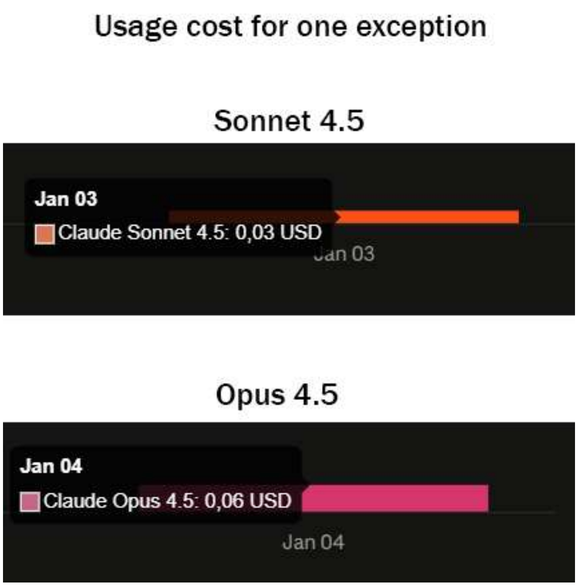
Since we track state locally, we need to know when PRs are closed. A simple GitHub webhook calls our agent when a PR is merged or rejected, clearing the cooldown cache so re-analysis can happen if needed.

The Cost: Is it Expensive?

I was actually agreeably surprised by how low the cost for the whole process is.

Claude API cost

I tried two models. Sonnet and Opus 4.5. No doubt Opus is the best but is also more expensive and not necessary needed for simple tasks.

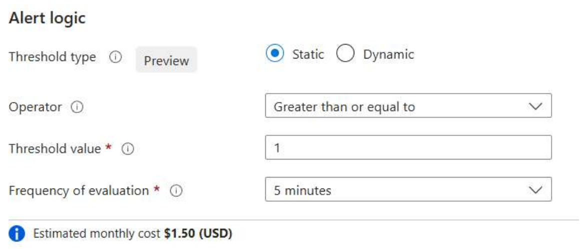


For minimizing cost we can use Claude Sonnet 4.5 model.

Approximately 0,03$ per simple exception (first prompt context is enough).
For more consistence or budget is not a priority, simply go ahead with Opus 4.5, which is around 0,06$ for an exception.

Alert rule cost:

As our alert rule is constantly running a query every 5 mins, the cost has to be considered too:



The estimated monthly cost if you run it 24h/7 is approximately $1.50.

Agent Hosting

If you plan using Azure to deploy your agent, most of the time using a Container App would cost nothing.

If you use a free container registry (such as Docker Hub or GitHub Container Registry) you would have no other fees.
If you stay on Azure then using the Azure Container Registry would cost $5.00/month for Basic tier.

As you see, that's less than $10/month total for an always-on automated diagnostics system.

What's Next?

The Agent is a prototype still, there is few enhancements that could be added to be production ready.

1. Multi-Agent (Coder Agent, Reviewer Agent..)
2. Rework on Pull Request comments (Agent would listen to your comments and suggest new fix)
3. Slack/Teams Integration to notify when PRs are created Allow manual approval before creation
4. Learning Loop Track which fixes were merged vs. rejected Fine-tune the prompt based on feedback

And more.

Conclusion

We built a complete self-healing system that:

- Monitors 24/7 via Azure Monitor
- Alert Rules Triggers automatically via webhooks
- Filters non-actionable exceptions (3-level smart filtering)
- Deduplicates at multiple levels (Redis + existing PR check)
- Analyzes with Claude in optimized LLM calls
- Creates Pull Requests automatically
- Costs ~$0.04 per analysis

Resources

- GitHub Repo (Source Code & Bicep)
- Microsoft Semantic Kernel Documentation

. . .

Did you enjoy this deep dive? Follow me here or on LinkedIn for more content.