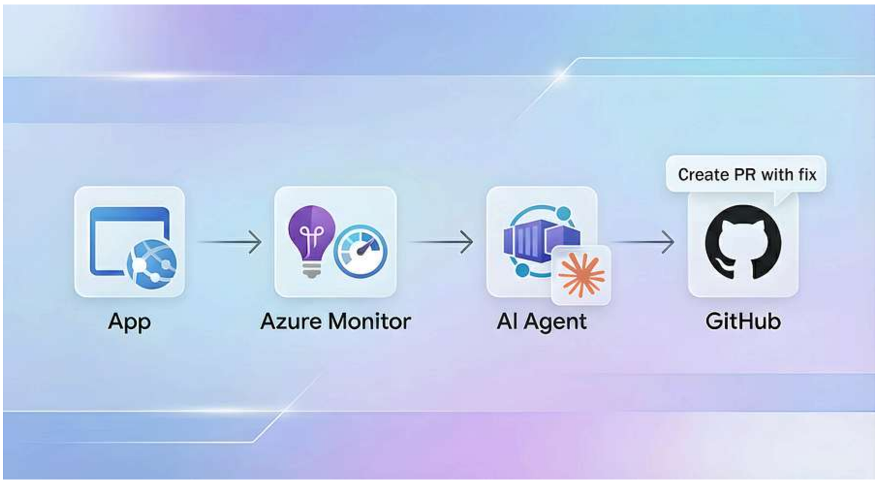
From Exceptions to Pull Requests: Building a Self Fixing .NET 10 App with AI Agents

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.

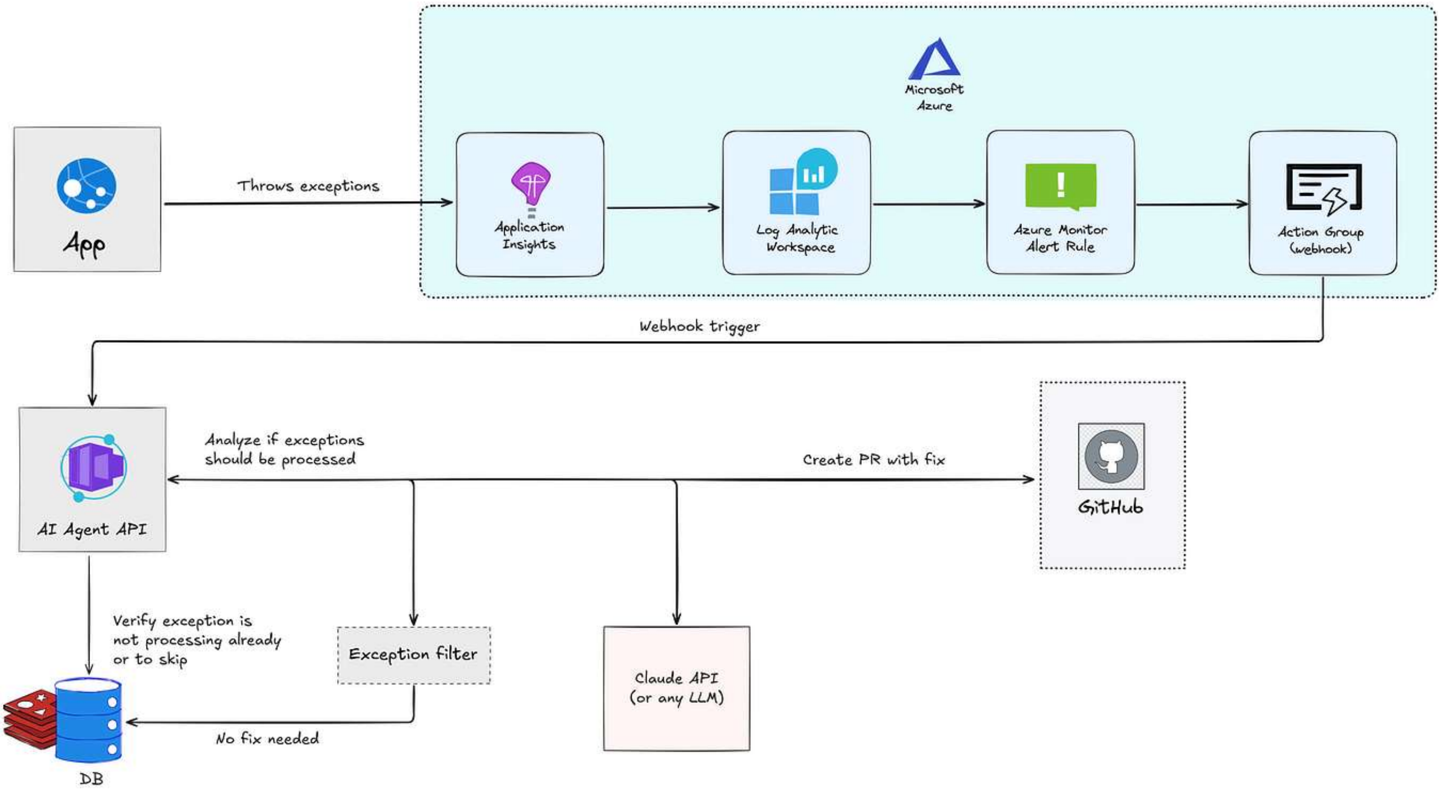
- **Chatbot:** "Here is how you might fix that `NullReferenceException`..."
- **Agent:** Reads the code, identifies the null object, creates a branch, commits the fix, and opens a Pull Request.

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:

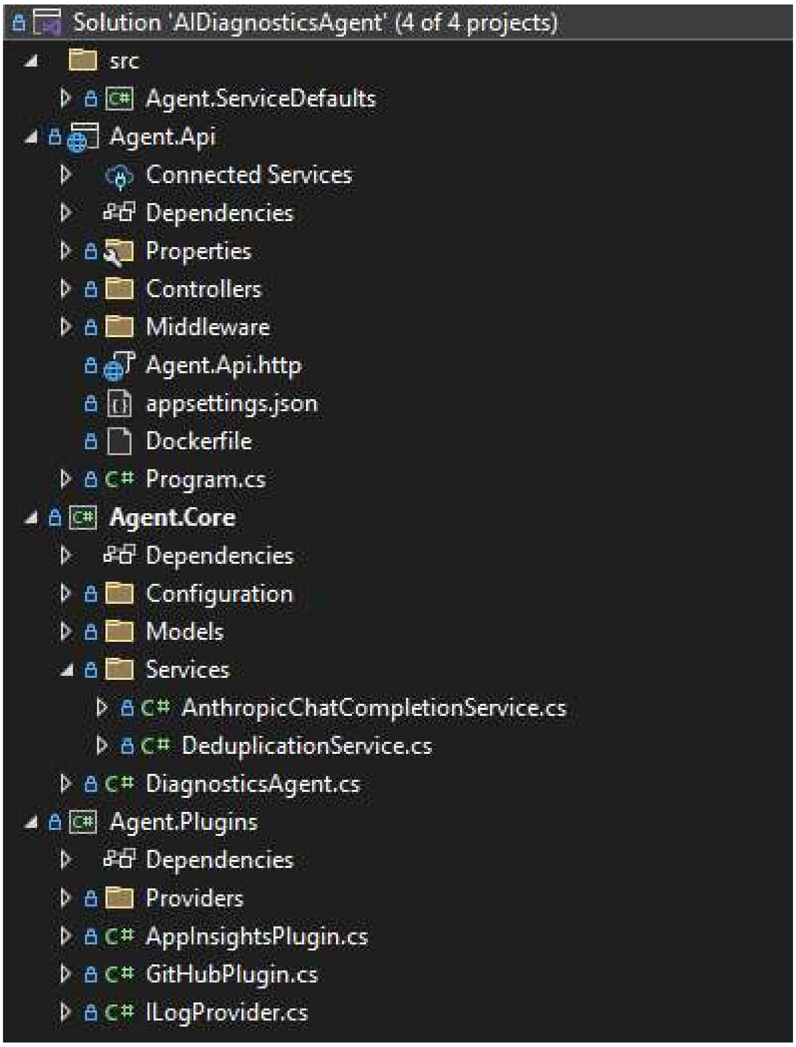
1. **The App** throws an exception.
2. **Azure Monitor** detects it and triggers an Alert Rule.
3. **The Agent API** receives a webhook, analyzes the issue, and consults the LLM (Claude Opus 4.5/Sonnet 4.5).
4. **GitHub** receives a new Pull Request if a fix is found.

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:



```
Solution 'AIDiagnosticsAgent' (4 of 4 projects)
  src
    Agent.ServiceDefaults
  Agent.Api
    Connected Services
    Dependencies
    Properties
    Controllers
    Middleware
    Agent.Api.http
    appsettings.json
    Dockerfile
    Program.cs
  Agent.Core
    Dependencies
    Configuration
    Models
    Services
      AnthropicChatCompletionService.cs
      DeduplicationService.cs
    DiagnosticsAgent.cs
  Agent.Plugins
    Dependencies
    Providers
    AppInsightsPlugin.cs
    GitHubPlugin.cs
    ILogProvider.cs
```

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

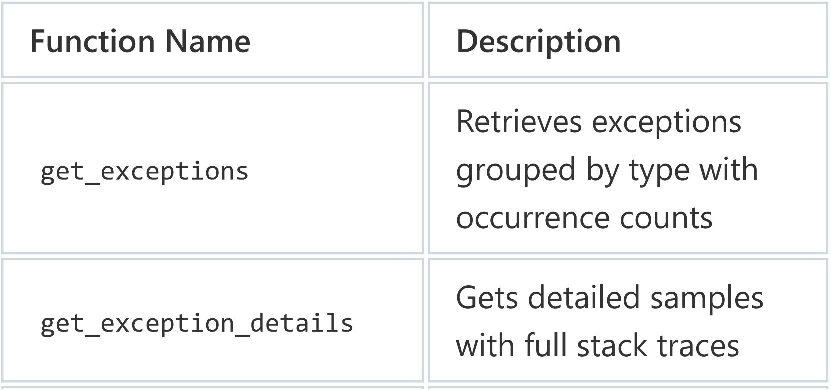
The brain of the operation. This is where the orchestration happens, among other things:

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

Agent.Plugins

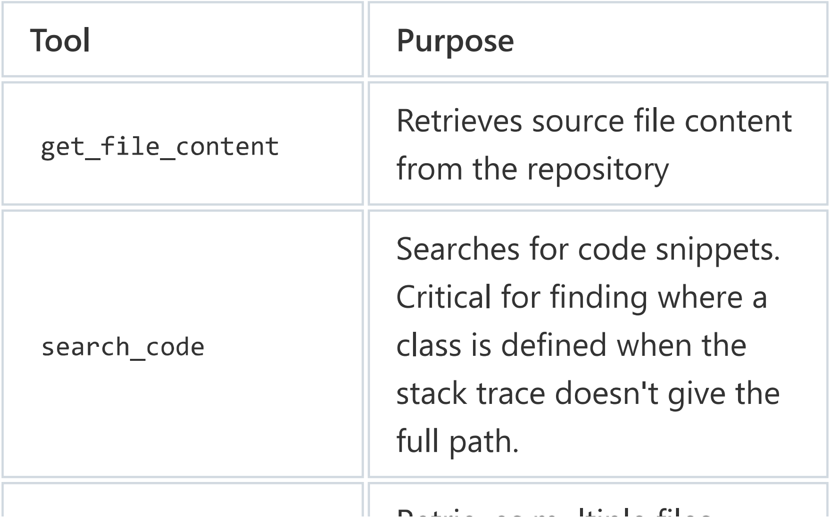
The custom tools that Claude can invoke. I set up a list of tools for communicating with Azure AppInsights and GitHub. Find the list below.

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.
	Retrieves multiple files

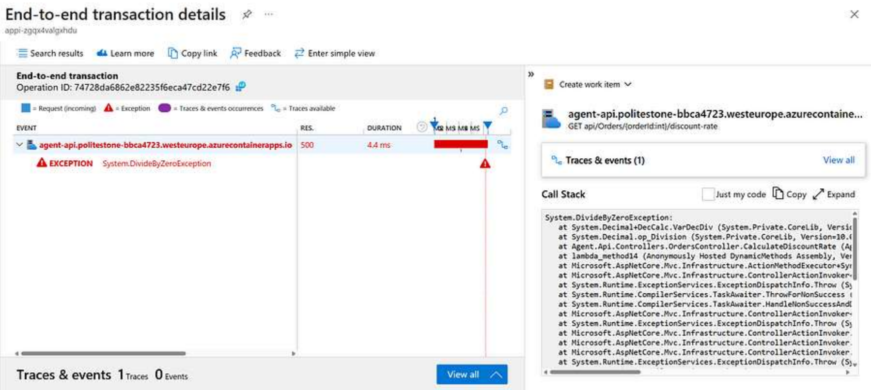
These plugins are registered with Semantic Kernel, allowing the LLM to understand when and how to use them based on the context.

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.


End-to-end transaction details

appi-zgqx4valgxhdu

End-to-end transaction
Operation ID: 74728da6862e82235f6eca47cd22e7f6

EVENT	RES.	DURATION
agent-api.politestone-bbca4723.westeurope.azurecontainerapps.io	500	4.4 ms
EXCEPTION System.DivideByZeroException		

Traces & events 1 Traces 0 Events

agent-api.politestone-bbca4723.westeurope.azurecontaine...
GET api/Orders/{orderId:int}/discount-rate

Traces & events (1)

Call Stack

```
System.DivideByZeroException:
   at System.Decimal+DecCalc.VarDecDiv (System.Private.CoreLib, Versi
   at System.Decimal.op_Division (System.Private.CoreLib, Version=10.
   at Agent.Api.Controllers.OrdersController.CalculateDiscountRate (A
   at lambda_method14 (Anonymously Hosted DynamicMethods Assembly, Ve
   at Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor+Sy
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker
   at System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw (S
   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess 
   at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAnd
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker
   at System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw (S
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
   at System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw (S
```

App Insights interface on exception details (DivideByZeroException).

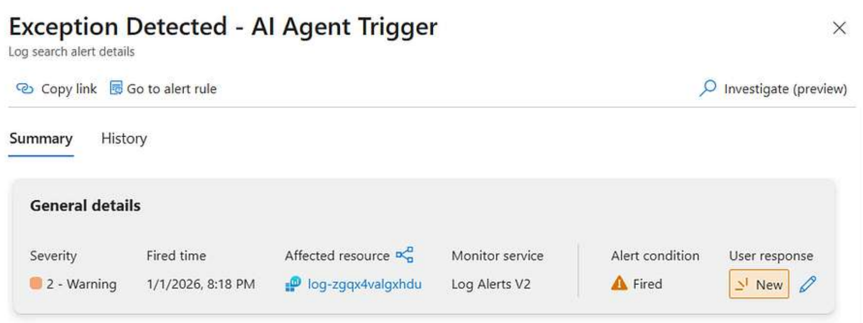
Step 1: The Trigger (Azure Monitor)

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:



Exception Detected - AI Agent Trigger

Log search alert details

General details

Severity	Fired time	Affected resource	Monitor service	Alert condition	User response
2 - Warning	1/1/2026, 8:18 PM	log-zgqx4valgxhdu	Log Alerts V2	Fired	New

Handling Deduplication

Without deduplication, the same NullReferenceException thrown 50 times would trigger 50 Claude API calls and 50 identical PRs. Not ideal, right?

To solve this, we use two layers of protection.

1. **Immediate Deduplication**

In a production environment, we cannot afford to trigger 1,000 LLM analysis calls for the same recurring bug. I implemented a “fingerprinting” strategy using Redis. Before calling the AI, the agent generates a unique hash based on the ExceptionType and ProblemId.

```
var exceptionHash = ComputeHash($"{evt.Type}_{ev

if (await _cache.ExistsAsync(exceptionHash))
{
    return; // Skip duplicate
}
await _cache.SetAsync(exceptionHash, true, TimeS
```

> *I used Redis for this demo because it is simple and fast to implement. However, for a production environment, I’d recommend adding a database to store fix history: which exceptions were fixed, and which PRs were merged vs. rejected.*

2. GitHub PR Check

Simply checking for recent alerts isn’t enough. If no open PR exists, it doesn’t mean we should re-analyze immediately. The exception might reoccur while the fix is being deployed, or the developer might still be reviewing the previous attempt. We enforce a cooldown before allowing re-analysis of the same problem to avoid spamming the repo.

Staying in Sync with GitHub

Since we track state locally (in Redis or DB), we need to know when PRs are actually closed. A simple GitHub webhook calls our agent when a PR is merged or rejected, clearing the cooldown cache so re-analysis can happen immediately if the bug persists.

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.

Step 3: Tool Execution (Semantic Kernel)

To interact with the outside world, we use **Microsoft Semantic Kernel**.

What is Semantic Kernel? It 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 it for three purposes:

- Let us swap between Claude, OpenAI, or Azure OpenAI with a simple config change.

- Plugin Registry
- Agent Orchestration: It handles the complex loop of “Thinking” and “Acting”.

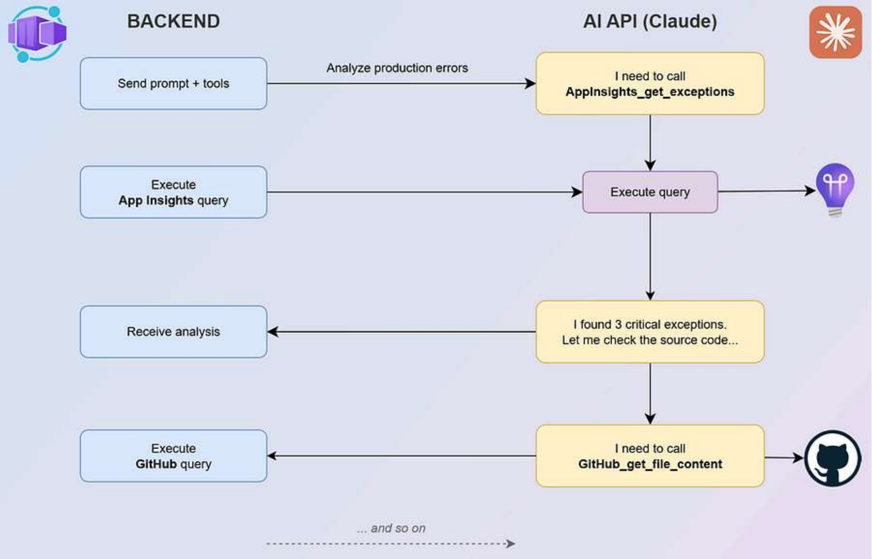
```
var agent = new ChatCompletionAgent
 {
   Name = "DiagnosticsAgent",
   Instructions = SystemPrompt,
   Kernel = kernel, // Contains all registered p
   Arguments = new KernelArguments(new OpenAIPro
   {
   FunctionChoiceBehavior = FunctionChoiceBehavi
   })
 };
```

By enabling `FunctionChoiceBehavior.Auto()` , the LLM autonomously decides: "I see the stack trace points *to* `OrderService.cs` *,* but I don't have the file content. I will call `get_file_content` to read it."

The GitHub Plugin: We define “Plugins” as simple C# functions decorated with `[KernelFunction]` .

```
[KernelFunction("get_file_content")]
[Description("Retrieves source file content from
public async Task<string> GetFileContent(string 
{
    // GitHub API implementation to fetch raw co
}
```

How Tool Use Actually Works



Agent Flow Diagram — showing the loop between LLM, Kernel, and Tools

When building an AI agent, simply telling Claude about available tools isn't enough. Here is what actually happens behind the scenes:

1. When calling the Claude API, we pass a `tools` array (JSON schemas). This tells the LLM strictly *what* exists and *how* to call it.

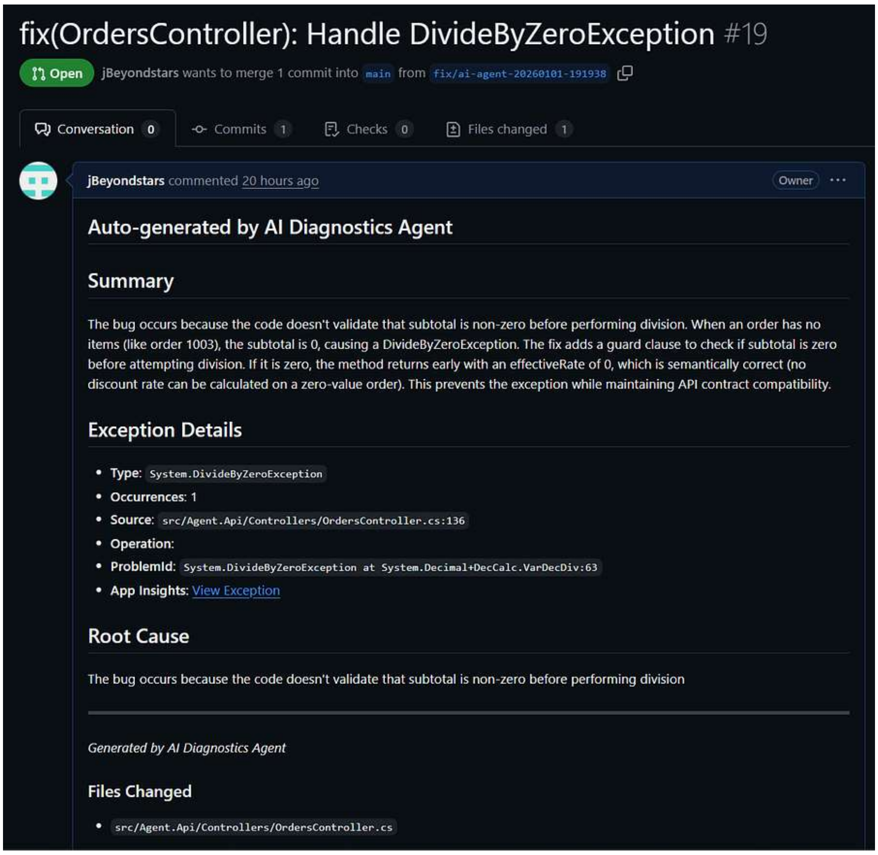
2. We inject the Kernel into the Agent.

3. The execution loop (that we want to minimize for cost and latency reduction):

1. Claude send a `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

Step 4: The Fix (Pull Request)

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



fix(OrdersController): Handle DivideByZeroException #19

Open jBeyondstars wants to merge 1 commit into `main` from `fix/ai-agent-20260101-191938`

Conversation 0 · Commits 1 · Checks 0 · Files changed 1

jBeyondstars commented 20 hours ago · Owner

Auto-generated by AI Diagnostics Agent

Summary

The bug occurs because the code doesn't validate that subtotal is non-zero before performing division. When an order has no items (like order 1003), the subtotal is 0, causing a DivideByZeroException. The fix adds a guard clause to check if subtotal is zero before attempting division. If it is zero, the method returns early with an effectiveRate of 0, which is semantically correct (no discount rate can be calculated on a zero-value order). This prevents the exception while maintaining API contract compatibility.

Exception Details

- **Type:** `System.DivideByZeroException`
- **Occurrences:** 1
- **Source:** `src/Agent.Api/Controllers/OrdersController.cs:136`
- **Operation:**
- **ProblemId:** `System.DivideByZeroException at System.Decimal+DecCalc.VarDecDiv:63`
- **App Insights:** View Exception

Root Cause

The bug occurs because the code doesn't validate that subtotal is non-zero before performing division

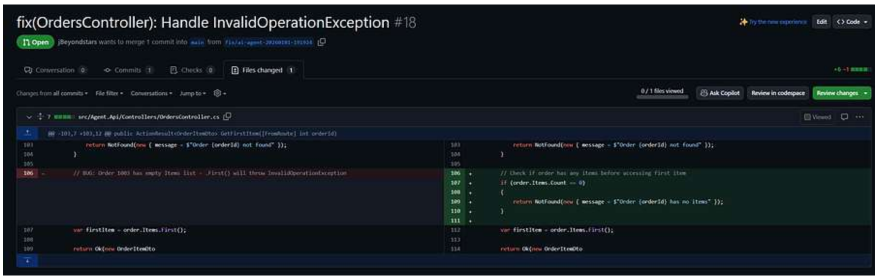
Generated by AI Diagnostics Agent

Files Changed

- `src/Agent.Api/Controllers/OrdersController.cs`

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.



fix(OrdersController): Handle InvalidOperationException #18

Open jBeyondstars wants to merge 1 commit into `main` from `fix/ai-agent-20260101-191924`

Conversation 0 · Commits 1 · Checks 0 · Files changed 1

`src/Agent.Api/Controllers/OrdersController.cs` +6 −1

```
@@ -103,7 +103,12 @@ public ActionResult<OrderItemDto> GetFirstItem([FromRoute] int orderId)
            return NotFound(new { message = $"Order {orderId} not found" });
        }

-       // BUG: Order 1003 has empty Items list - .First() will throw InvalidOperationException
+       // Check if order has any items before accessing first item
+       if (order.Items.Count == 0)
+       {
+           return NotFound(new { message = $"Order {orderId} has no items" });
+       }
+
        var firstItem = order.Items.First();

        return Ok(new OrderItemDto
```

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: Hardcoded exceptions type exclusion

```
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: Partial String Matching

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

Level 3: Claude Evaluation

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

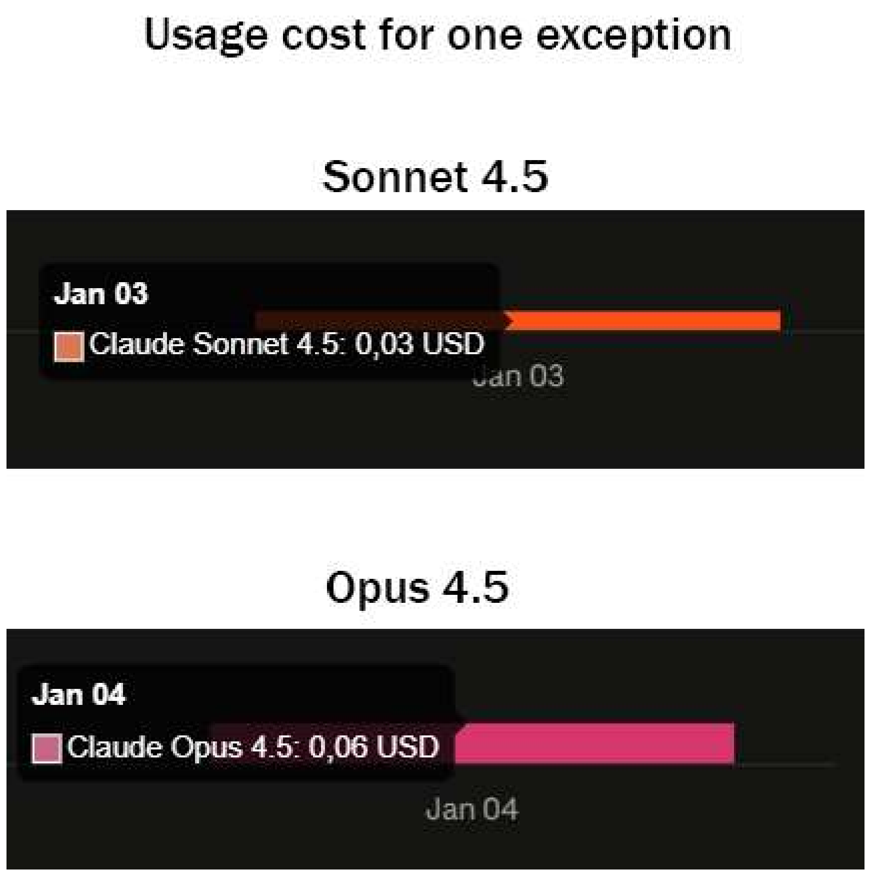
```
private async Task<bool> EvaluateIfFixableAsync(
{
var prompt =
"You are evaluating if a production exception ca

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";
// ..
}
```

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.



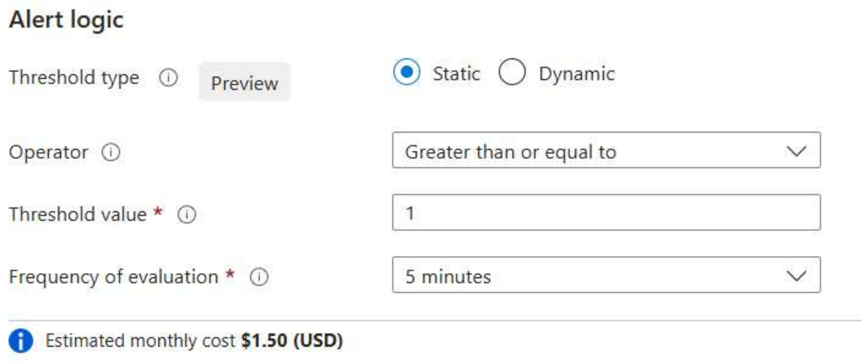
Usage cost for one exception

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:



Alert logic

Threshold type ⓘ Preview ● Static ○ Dynamic

Operator ⓘ Greater than or equal to

Threshold value * ⓘ 1

Frequency of evaluation * ⓘ 5 minutes

ⓘ Estimated monthly cost **$1.50 (USD)**

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.