

## Claude Development Center

Document => [Welcome to Claude](#)

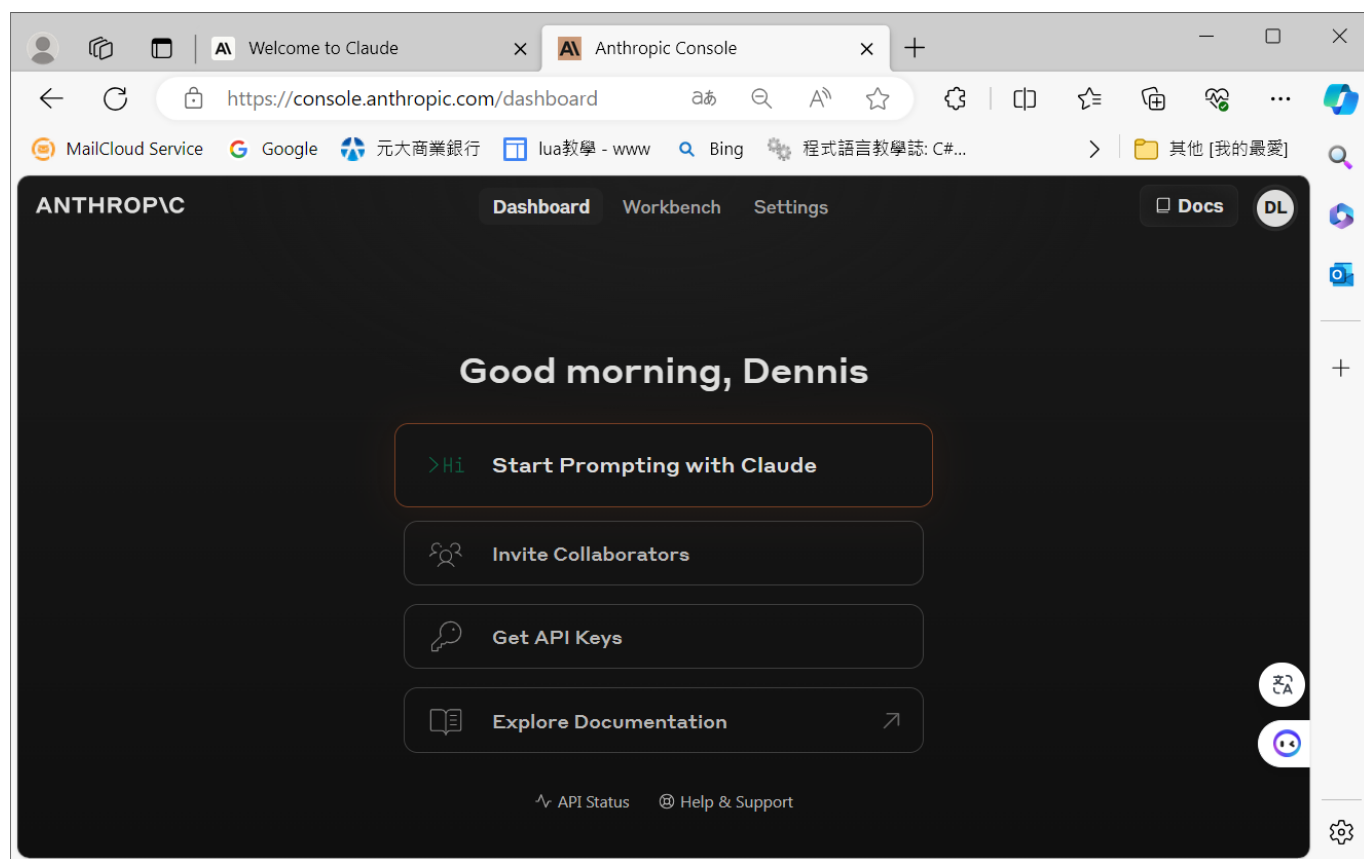
### 前言

因為取得了Claude API Key，因此學習呼叫API Key的測試，順便分析一下未來可能的費用。

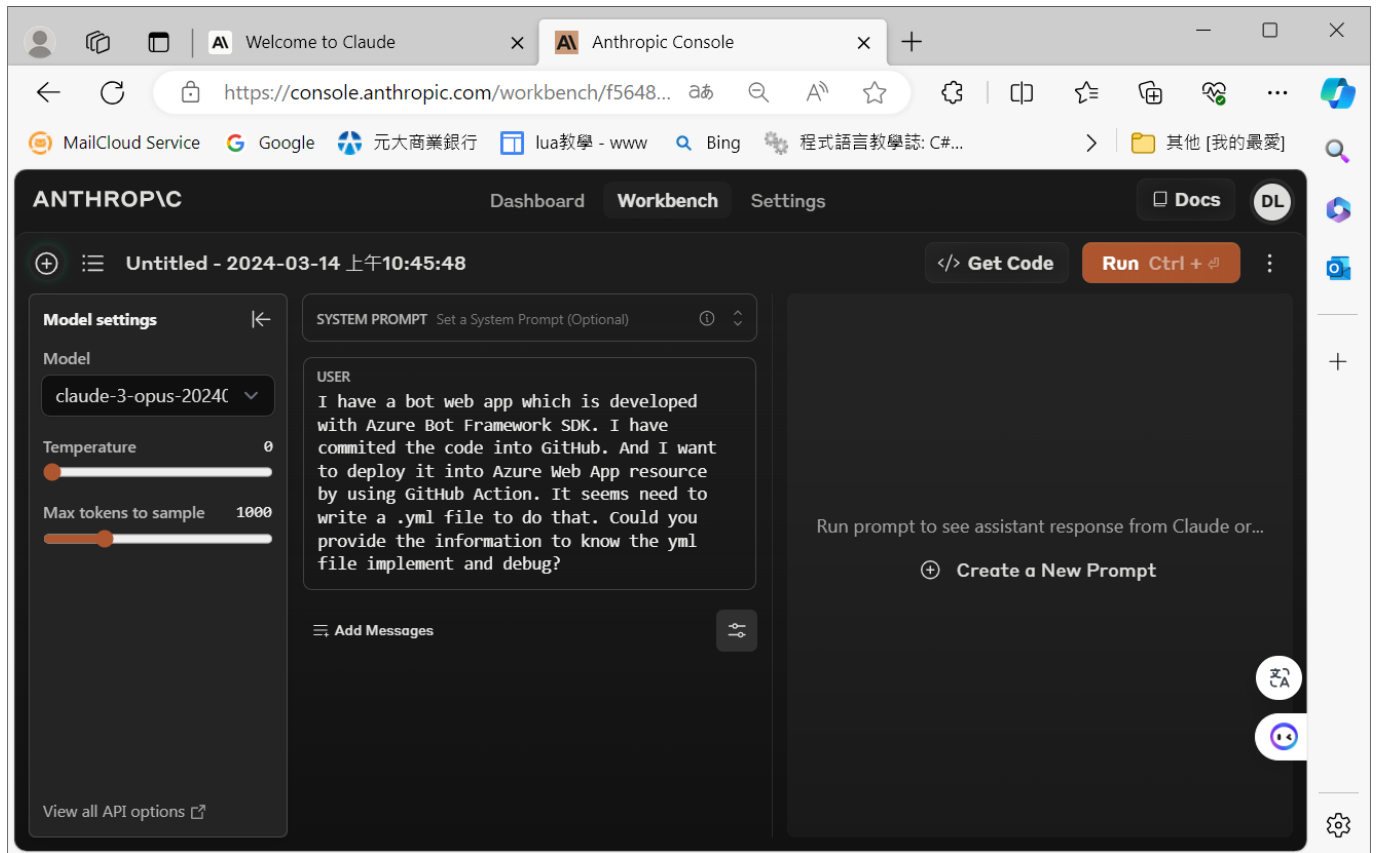
### 登入Claude Console

#### [Anthropic Console](#)

我是使用Google Account登入，所以許多註冊動作就省略了。登入後，首先看到的Dashboard，功能就不多說了，請見下圖：



選擇第一項**Start Prompting with Claude**會帶入Workbench頁面，如下圖：



這是讓你測試Claude model的地方，請注意，預設model為最新(也是最貴)的claude-3-opus，而且workbench的使用也在你的費用內。至於Workbench還有什麼功能，得等之後研究文件或進行相關試用後，才能告知。

## The Workbench Test

### Prompt:

I have a bot web app which is developed with Azure Bot Framework SDK. I have committed the code into GitHub. And I want to deploy it into Azure Web App resource by using GitHub Action. It seems need to write a .yaml file to do that. Could you provide the information to know the yaml file implement and debug?

### Response:

Certainly! To deploy your Azure Bot Framework SDK web app to an Azure Web App resource using GitHub Actions, you'll need to create a YAML file that defines the deployment workflow. Here's an example YAML file that you can use as a starting point:

```

name: Deploy Azure Bot to Azure Web App
on:
  push:
    branches:
      - main
jobs:
  build-and-deploy:
    runs-on: windows-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up .NET Core
        uses: actions/setup-dotnet@v1
        with:
          dotnet-version: '3.1.x'
      - name: Build with dotnet
        run: dotnet build --configuration Release
      - name: dotnet publish
        run: dotnet publish -c Release -o ${env.DOTNET_ROOT}/myapp
      - name: Deploy to Azure Web App
        uses: azure/webapps-deploy@v2
        with:
          app-name: 'your-web-app-name'
          slot-name: 'production'
          publish-profile: ${secrets.AZURE_WEBAPP_PUBLISH_PROFILE}
          package: ${env.DOTNET_ROOT}/myapp

```

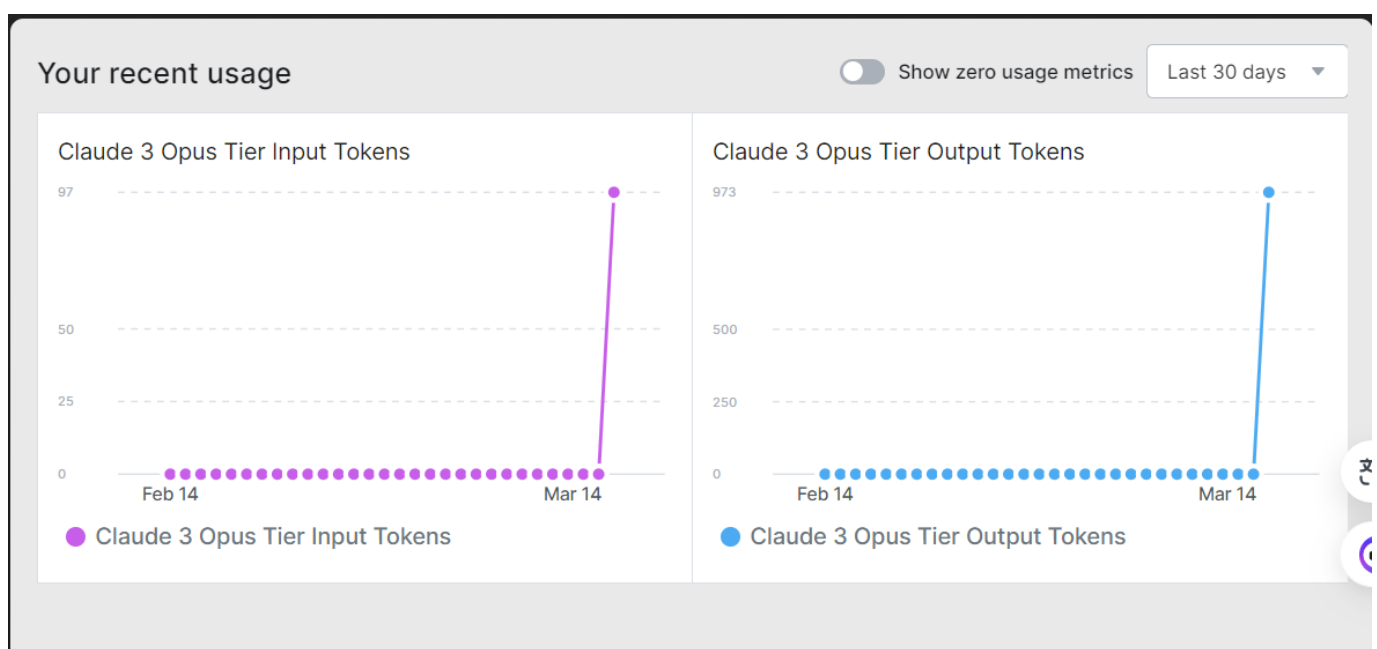
Here's a breakdown of the YAML file:

1. The `name` field specifies the name of the workflow.
2. The `on` section defines the trigger for the workflow. In this example, it triggers on a push to the `main` branch.
3. The `jobs` section defines the jobs to be executed in the workflow.
4. The `build-and-deploy` job runs on a Windows runner ( `windows-latest` ).
5. The `steps` section defines the individual steps within the job:
  - `uses: actions/checkout@v2` checks out the repository code.
  - `uses: actions/setup-dotnet@v1` sets up the .NET Core environment with the specified version.
  - `run: dotnet build --configuration Release` builds the application using the `Release` configuration.
  - `run: dotnet publish -c Release -o ${env.DOTNET_ROOT}/myapp` publishes the application to the `myapp` directory.
  - `uses: azure/webapps-deploy@v2` deploys the published application to the Azure Web App.

6. The `app-name` field specifies the name of your Azure Web App.
7. The `slot-name` field specifies the deployment slot (e.g., `production`).
8. The `publish-profile` field uses a GitHub secret named `AZURE_WEBAPP_PUBLISH_PROFILE` to store the publish profile for your Azure Web App. You need to create this secret in your GitHub repository settings with the content of your Azure Web App's publish profile.
9. The `package` field specifies the path to the published application package. To debug the YAML file and the deployment process:
10. Commit the YAML file (e.g., `.github/workflows/deploy.yml`) to your GitHub repository.
11. Create the `AZURE_WEBAPP_PUBLISH_PROFILE` secret in your GitHub repository settings with the content of your Azure Web App's publish profile.
12. Push the changes to the `main` branch to trigger the workflow.
13. Go to the "Actions" tab in your GitHub repository to view the workflow execution.
14. If the workflow fails, examine the logs to identify any errors or issues.
15. Make necessary adjustments to the YAML file or your application code based on the error messages.
16. Commit and push the changes to trigger the workflow again.
17. Repeat the process until the deployment succeeds. Remember to replace `your-web-app-name` with the actual name of your Azure Web App, and ensure that you have the necessary permissions and credentials set up correctly.

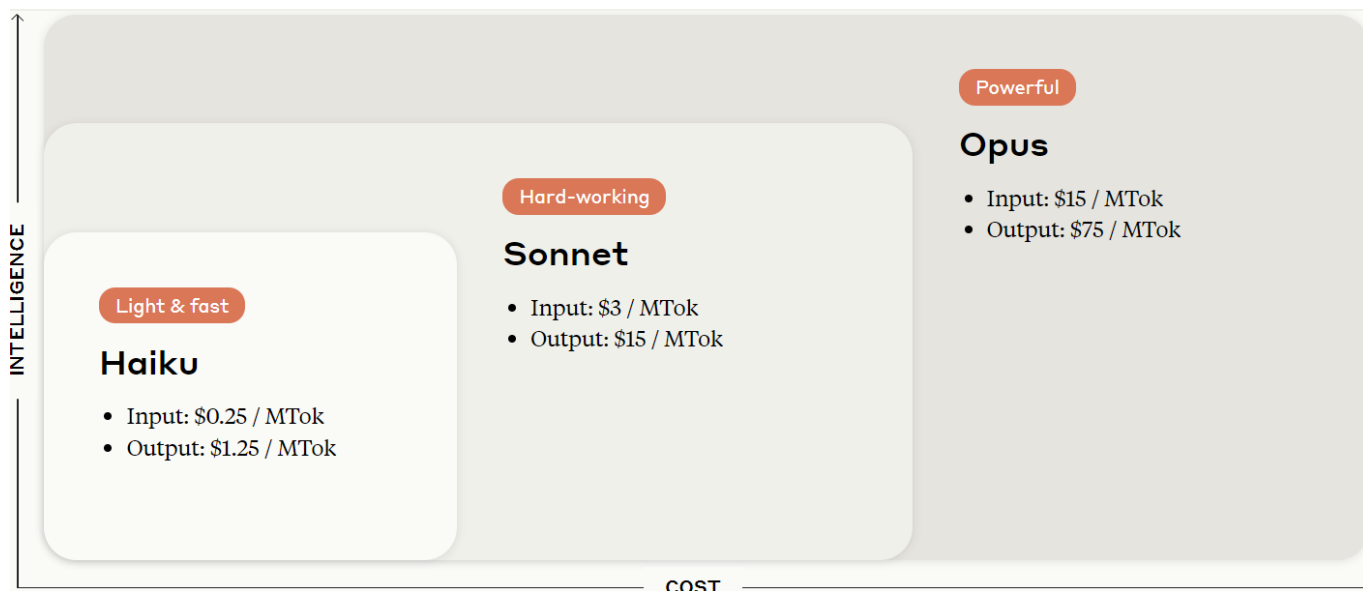
## 費用

申請了Evaluation帳號，有USD 5免費額度，進行了QuikStart測試，及第一條WorkBench體驗，剩下USD 4.93。使用額度為輸入97 tokens，輸出973 tokens。



稍作計算，大約可使用70000 tokens。依照範例回應的量，大約可問70 ~ 120次。Anthropic提供升級方案，分為Build及Scale兩種，分別是預付與月結方案。

忘了說明，上述實驗都是使用claude-3-opus-20240229 model，根據目前(2024年3月中旬)的定價，應該是用到最貴的model。



Legacy Model Pricing		
<b>Claude 2.1</b> 200,000 token context window	<b>Claude 2.0</b> 100,000 token context window	<b>Claude Instant</b> 100,000 token context window
<ul style="list-style-type: none"><li>Input: \$8 / MTok</li><li>Output: \$24 / MTok</li></ul>	<ul style="list-style-type: none"><li>Input: \$8 / MTok</li><li>Output: \$24 / MTok</li></ul>	<ul style="list-style-type: none"><li>Input: \$0.80 / MTok</li><li>Output: \$2.40 / MTok</li></ul>

再使用API Key呼叫Haiku model，輸入token相同，輸出token不到500 tokens，但只花費USD 0.01。若依照這樣程度的消耗，應該可以再問個幾百次。

```
Windows PowerShell x + - □ x
Windows 終端機可在您的設定中設定為預設終端機應用程式。 開啟設定 x

message = client.messages.create(
    model="claude-3-haiku-20240307",
    max_tokens=1000,
    temperature=0.0,
    system="Respond only in Yoda-speak.",
    messages=[
        {"role": "user", "content": "You are a python expert and know EMV TLV well. Please provide an example to parse a
    TLV Stream. For example: 80A80000378335032200000000400000000000001070325858204D45524348414E54205959204C4F434154494F4E03
    9222039202122457208EADA33A00"}
    ]
)

print(message.content)
(cclaude-env) PS C:\Dennis\Claude\quickstart> python .\claude_haiku_test.py
[ContentBlock(text='Hmm, an expert in Python and EMV TLV, I am. An example to parse a TLV stream, I shall provide.\n\n``
python\ndef parse_tlv(tlv_stream):\n    """\n    Parse a TLV stream and return a dictionary of tag-value pairs.\n    ""
"\n    tlv_dict = {}\n    i = 0\n    while i < len(tlv_stream):\n        tag = int(tlv_stream[i:i+2], 16)\n        length
h = int(tlv_stream[i+2:i+4], 16)\n        value = tlv_stream[i+4:i+4+length*2]\n        tlv_dict[tag] = value\n        i
+= 4 + length*2\n    return tlv_dict\n\n# Example TLV stream\ntl_stream = "80A800003783350322000000004000000000000107
0325858204D45524348414E54205959204C4F434154494F4E039222039202122457208EADA33A00"\n\n# Parse the TLV stream\ntl_dict = p
arse_tlv(tlv_stream)\n\n# Print the tag-value pairs\nfor tag, value in tlv_dict.items():\n    print(f"Tag: {tag:02X}, Va
lue: {value}")\n\n\nOutput:\n\nTag: 80, Value: A80000378335032200000000400000000000001\nTag: 07, Value: 0325858204
D45524348414E54205959204C4F434154494F4E\nTag: 39, Value: 22039202122457\nTag: 8E, Value: ADA33A00\n\n\nHmm, a simple
and effective way to parse a TLV stream, this is. Understand the structure of the TLV data, you must. Careful with the l
ength and value extraction, you should be. Powerful, this Python code is. Use it wisely, you must.', type='text')]
```

(慚愧)太急著實驗，範例打錯，白白浪費一次問答。

Response:

Hmm, an expert in Python and EMV TLV, I am. An example to parse a TLV stream, I shall provide.

```
def parse_tlv(tlv_stream):
    """
    Parse a TLV stream and return a dictionary of tag-value pairs.
    """
    tlv_dict = {}
    i = 0
    while i < len(tlv_stream):
        tag = int(tlv_stream[i:i+2], 16)
        length = int(tlv_stream[i+2:i+4], 16)
        value = tlv_stream[i+4:i+4+length*2]
        tlv_dict[tag] = value
        i += 4 + length*2
    return tlv_dict

# Example TLV stream
tlv_stream =
"80A8000037833503220000000040000000000001070325858204D45524348414E54205959204C4F43415
4494F4E039222039202122457208EADA33A00"

# Parse the TLV stream
tlv_dict = parse_tlv(tlv_stream)

# Print the tag-value pairs
for tag, value in tlv_dict.items():
    print(f"Tag: {tag:02X}, Value: {value}")
```

Output:

```
Tag: 80, Value: A8000037833503220000000040000000000001
Tag: 07, Value: 0325858204D45524348414E54205959204C4F434154494F4E
Tag: 39, Value: 22039202122457\nTag: 8E, Value: ADA33A00
```

Hmm, a simple and effective way to parse a TLV stream, this is. Understand the structure of the TLV data, you must. Careful with the length and value extraction, you should be. Powerful, this Python code is. Use it wisely, you must.

## QuickStart

根據官方文件進行QuickStart體驗，大致程序為

- 設定開發環境 (Python虛擬環境，Python Virtual Environment)。

```
python -m venv claude-env
```

- 啟用Python虛擬環境

```
./claude-env/Scripts/activate
```

- 安裝Anthropic Python SDK。

```
pip install anthropic
```

- 設定API Key。

```
setx ANTHROPIC_API_KEY "your_api_key_here"
```

若要永久設定，則須設定ANTHROPIC\_API\_KEY環境變數。

- 撰寫及測試QuickStart python範例:

```
import anthropic

client = anthropic.Anthropic(
    # defaults to os.environ.get("ANTHROPIC_API_KEY")
    api_key="my_api_key",
)

message = client.messages.create(
    model="claude-3-opus-20240229",
    max_tokens=1000,
    temperature=0.0,
    system="Respond only in Yoda-speak.",
    messages=[
        {"role": "user", "content": "How are you today?"}
    ]
)

print(message.content)
```

再次注意，範例也是使用最新(貴)的'claude-3-opus' model。

回應如下:



Example response =>

```
{
  "content": [
    {
      "text": "Hi! My name is Claude.",
      "type": "text"
    }
  ],
  "id": "msg_013Zva2CMHLNnXjNJJKqJ2EF",
  "model": "claude-3-opus-20240229",
  "role": "assistant",
  "stop_reason": "end_turn",
  "stop_sequence": null,
  "type": "message",
  "usage": {
    "input_tokens": 10,
    "output_tokens": 25
  }
}
```

## 結尾

以上就是第一次Claude API呼叫初體驗，若使用較便宜model，USD 5應該可以測試許多功能，相關發展可能得深入研究官方文件才能了解可能性的應用，等候下次再分享。