You've hit on a crucial aspect of enterprise-scale AI adoption: **self-service onboarding** and **integration with existing workflows**. The previous code module provided the *infrastructure* for the AI Hub and its gateway. Now, let's discuss how to enable the self-service aspect for application teams and integrate it with Azure DevOps work items.

Self-Service Onboarding with Azure Pipelines & Azure DevOps

The concept of "self-service" for application teams typically involves:

1. **A request mechanism:** How does a team ask for access?
2. **An approval workflow:** Who reviews and approves the request?
3. **Automated provisioning:** What happens after approval?

1. Request Mechanism (Azure Boards Work Items)

This is where Azure Boards can play a key role. Application teams can raise a work item (e.g., a "New AI Hub Access Request" task or a custom work item type) in Azure Boards.
Work Item Fields: Your custom work item type might include fields like:

- **Application Name:** (e.g., "Customer Sentiment Analysis App")
- **Team Name:** (e.g., "Marketing Analytics")
- **Required AI Models:** (e.g., "GPT-4", "Text-to-Speech")
- **Expected Usage (RPM/TPM):** (Requests per minute, Tokens per minute)
- **Business Justification:**
- **Requested APIM Product:** (e.g., "OpenAI Model Access" - as provisioned in the previous module)
- **Contact Email:** For the application team.
- **Status:** (New, Approved, Rejected, Provisioning, Provisioned, Failed) - This is key for triggering.

2. Approval Workflow (Azure DevOps Environments & Work Item State)

Azure DevOps Pipelines have built-in approval mechanisms through **Environments**. For the work item triggering, you'd combine this with a custom webhook.
How to set this up:

1. **Define an Azure DevOps Environment:**
    - Go to **Pipelines** -> **Environments**.
    - Create an Environment (e.g., AIHub_Onboarding_Approvals).
    - Add "Approvals" and "Checks" to this environment.
        - **Approvals:** Specify the team members (e.g., "AI Platform Team Lead", "Security Team") who must approve any deployment to this environment.
        - **Invoke Azure Function/REST API Check (Optional but powerful):** You could also add a check that calls an Azure Function or a REST API to verify additional business rules or even to update the work item state *before* the approval is granted.
2. **Pipeline Triggering (Work Item State Change):** This is not a direct built-in trigger in Azure DevOps. You'll need an intermediary:
    - **Workaround: Webhook + Azure Function / Logic App:**
        - **Azure Boards Webhook:** Configure a webhook in your Azure DevOps Project Settings -> Service Hooks. Set it to trigger when a work item of your

custom type (e.g., "New AI Hub Access Request") changes its Status field to Approved.
        - **Target URL:** This webhook will post a JSON payload to an Azure Function or Logic App HTTP trigger.
        - **Azure Function/Logic App Logic:**
            - Receive the webhook payload (which contains details of the work item, including its ID).
            - Parse the payload to extract the work item ID and other relevant details (application name, requested APIM product, etc.).
            - **Trigger the Azure DevOps Pipeline:** Use the Azure DevOps REST API (specifically, the "Runs - Run Pipeline" API) to trigger your "Application Onboarding" pipeline. Pass the work item details (e.g., application name, APIM product) as pipeline variables.
            - **Update Work Item Status:** After successfully triggering the pipeline, you could use the Azure DevOps REST API to update the work item status to "Provisioning".
            - **Error Handling:** Implement robust error handling if the pipeline trigger fails.
    - **Simpler Alternative (Manual Trigger with Pre-checks):** While not fully automated by work item state, a simpler approach is for the requesting team to manually trigger an "Onboarding" pipeline. The pipeline then has a pre-stage where it *checks* the work item status (e.g., using an Azure DevOps REST API task) and waits for it to be "Approved" before proceeding to the provisioning steps. This is less "dynamic" but simpler to implement.

3. Automated Provisioning (Separate Azure Pipeline)

You would create a **separate Azure Pipeline** specifically for application onboarding. This pipeline is much lighter than your infrastructure pipeline and focuses on API Management user/subscription creation.

azure-pipelines-app-onboarding.yml (Example)

```
# azure-pipelines-app-onboarding.yml
# This pipeline is triggered by an external system (e.g., Azure
Function from Work Item webhook)
# or manually by a platform administrator.

trigger: none # This pipeline will be triggered externally or manually

parameters:
  - name: applicationName
    type: string
    displayName: 'Application Name'
    default: 'MyNewApplication'
  - name: teamEmail
    type: string
    displayName: 'Team Contact Email'
    default: 'appteam@example.com'
  - name: apimProductName
```

```
    type: string
    displayName: 'API Management Product Name'
    default: 'openai-access' # Must match a product provisioned by
your infrastructure pipeline
  - name: workItemId
    type: string
    displayName: 'Azure DevOps Work Item ID'
    default: '0' # For tracking purposes, optional

variables:
  - group: TerraformBackendVariables # Reuse backend variables if
needed for APIM updates
  - name: serviceConnection
    value: $(AZURE_SERVICE_CONNECTION)
  - name: apimServiceName
    value: 'prod-aihub-apim-prod' # Or dynamically determined based on
environment param

stages:
  - stage: CreateAPIMSubscription
    displayName: 'Create APIM Subscription for Application'
    jobs:
      - deployment: CreateSubscriptionJob
        displayName: 'Create Subscription'
        environment: 'AIHub_Onboarding_Approvals' # Link to your ADO
Environment for pre-checks/approvals
        pool:
          vmImage: 'ubuntu-latest'
        strategy:
          runOnce:
            preDeploy:
              steps:
                - task: AzureCLI@2
                  displayName: 'Azure Login'
                  inputs:
                    azureSubscription: $(serviceConnection)
                    scriptType: 'bash'
                    scriptLocation: 'inlineScript'
                    inlineScript: |
                      az account show
                      # Optional: Add a check here to ensure the work
item is 'Approved' if not using webhooks
                      # az boards work-item show --id ${{
parameters.workItemId }} --query "fields.'System.State'"
            deploy:
              steps:
                - task: AzureCLI@2
                  displayName: 'Create APIM User and Subscription'
```

```
                  inputs:
                    azureSubscription: $(serviceConnection)
                    scriptType: 'bash'
                    scriptLocation: 'inlineScript' |
                      echo "Creating APIM user and subscription for
${{ parameters.applicationName }}"

                      APIM_RG="rg-prod-aihub" # Replace with your APIM
Resource Group name
                      APIM_SERVICE_NAME="$(apimServiceName)"
                      APP_NAME_SAFE="$(echo '${{
parameters.applicationName }}' | tr -d '[:space:]' | tr '[:upper:]'
'[:lower:]')"
                      USER_EMAIL="${{ parameters.teamEmail }}"
                      PRODUCT_ID="${{ parameters.apimProductName }}"

                      # 1. Check if user already exists, create if not
                      USER_ID=$(az apim user list --resource-group
"$APIM_RG" --service-name "$APIM_SERVICE_NAME" --filter "email eq
'$USER_EMAIL'" --query "[0].id" -o tsv)
                      if [ -z "$USER_ID" ]; then
                          echo "User $USER_EMAIL not found, creating
new user..."
                          USER_ID=$(az apim user create \
                              --resource-group "$APIM_RG" \
                              --service-name "$APIM_SERVICE_NAME" \
                              --user-id "$APP_NAME_SAFE-user" \
                              --email "$USER_EMAIL" \
                              --first-name "${{
parameters.applicationName }}" \
                              --last-name "Team" \
                              --state "active" \
                              --query "id" -o tsv)
                          echo "Created user with ID: $USER_ID"
                      else
                          echo "User $USER_EMAIL already exists with
ID: $USER_ID"
                      fi

                      # Extract just the user ID part (e.g.,
/users/app-name-user)
                      USER_PATH_ID=$(echo "$USER_ID" | sed
's/.*\(\/users\/[^\/]*\)/\1/')

                      # 2. Create APIM Subscription
                      echo "Creating subscription for product
$PRODUCT_ID and user $USER_PATH_ID"
                      SUBSCRIPTION_ID=$(az apim product subscription
```

```
create \
                            --resource-group "$APIM_RG" \
                            --service-name "$APIM_SERVICE_NAME" \
                            --product-id "$PRODUCT_ID" \
                            --subscription-name "${{
parameters.applicationName }}-access" \
                            --user-id "$USER_PATH_ID" \
                            --scope "/products/$PRODUCT_ID" \
                            --query "id" -o tsv)

                        echo "Created subscription with ID:
$SUBSCRIPTION_ID"

                        # 3. Get subscription keys
                        KEYS=$(az apim product subscription show-keys \
                            --resource-group "$APIM_RG" \
                            --service-name "$APIM_SERVICE_NAME" \
                            --sid "$SUBSCRIPTION_ID" \
                            --query "{primaryKey:primaryKey,
secondaryKey:secondaryKey}" -o json)

                        PRIMARY_KEY=$(echo "$KEYS" | jq -r
'.primaryKey')
                        SECONDARY_KEY=$(echo "$KEYS" | jq -r
'.secondaryKey')

                        echo "##vso[task.setvariable
variable=APIM_PRIMARY_KEY;isOutput=true]$PRIMARY_KEY"
                        echo "##vso[task.setvariable
variable=APIM_SECONDARY_KEY;isOutput=true]$SECONDARY_KEY"
                        echo "APIM Primary Key: $PRIMARY_KEY" # For
demo, mask in prod!
                        echo "APIM Secondary Key: $SECONDARY_KEY" # For
demo, mask in prod!

                        echo "##vso[task.setvariable
variable=apimSubscriptionId]$SUBSCRIPTION_ID"
                        echo "##vso[task.setvariable
variable=apimUserId]$USER_ID"

                - task: AzureCLI@2
                  displayName: 'Update Work Item Status to Provisioned
(if workItemId is provided)'
                  condition: and(succeeded(), ne('${{
parameters.workItemId }}', '0'))
                  inputs:
                    azureSubscription: $(serviceConnection)
                    scriptType: 'bash'
```

```
                    scriptLocation: 'inlineScript'
                    inlineScript: |
                      WORK_ITEM_ID=${{ parameters.workItemId }}
                      SUBSCRIPTION_ID=$(apimSubscriptionId) # Access
variable from previous step
                      USER_ID=$(apimUserId) # Access variable from
previous step
                      PRIMARY_KEY=$(APIM_PRIMARY_KEY)

                      # Build description with details, *masking
sensitive keys*
                      DESCRIPTION_UPDATE="<br/>APIM Access
Provisioned:<br/>Subscription ID: $SUBSCRIPTION_ID<br/>User ID:
$USER_ID<br/>Primary Key: (Provided securely out-of-band)"

                      # Update work item status and add comments
                      az boards work-item update \
                          --id "$WORK_ITEM_ID" \
                          --fields "System.State=Provisioned" \
                          --description "$DESCRIPTION_UPDATE" \
                          --comment "APIM access successfully
provisioned. Please retrieve your keys securely. (Subscription ID:
$SUBSCRIPTION_ID)"

                - task: AzureCLI@2
                  displayName: 'Send Secure Notification (e.g.,
Teams/Email)'
                  # This would be an additional step to notify the
application team
                  # with their API keys securely, WITHOUT printing to
logs.
                  # This could involve calling another Azure Function
with the keys,
                  # which then sends an email/Teams message.
                  inputs:
                    azureSubscription: $(serviceConnection)
                    scriptType: 'bash'
                    scriptLocation: 'inlineScript'
                    inlineScript: |
                      # Placeholder for secure notification logic
                      # Use Azure Functions or Logic Apps for secure
delivery of keys.
                      echo "Secure notification process triggered to
send keys to ${{ parameters.teamEmail }}"
                      # Example: az functionapp invoke --name
MyNotifierFunc --function-name SendKeys --body "{'email':'${{
parameters.teamEmail }}', 'primaryKey':'$(APIM_PRIMARY_KEY)'}"
```

Key Components and Considerations:

1. **Terraform vs. Azure CLI for Onboarding:**
    - For azurerm_api_management_user and azurerm_api_management_subscription, you *could* use Terraform.
    - **However, for self-service onboarding, directly using Azure CLI (as shown in the azure-pipelines-app-onboarding.yml example) or Azure PowerShell within the pipeline is often preferred.**
        - **Reason 1: Idempotency & State:** Managing hundreds or thousands of individual APIM subscriptions with a single Terraform state file can become unwieldy and slow. Each subscription is a separate logical unit.
        - **Reason 2: Dynamic Nature:** These are operational tasks that happen frequently and are driven by user requests, not core infrastructure changes. A CLI script handles this more dynamically.
        - **Reason 3: Simpler Key Retrieval:** Retrieving the generated subscription keys (az apim product subscription show-keys) is straightforward with CLI/PowerShell, whereas exposing them as Terraform outputs for thousands of subscriptions isn't practical.
2. **Secure Key Delivery:**
    - **NEVER log API keys directly in the pipeline output.** The example above uses isOutput=true but in a real scenario, you'd immediately pass these to a secure mechanism.
    - **Recommended:** After generating keys in the pipeline, call an Azure Function or Logic App that is responsible for:
        - Encrypting the keys.
        - Storing them securely (e.g., in Azure Key Vault, linked to the application's service principal for access).
        - Notifying the application team via a secure channel (e.g., direct email with a link to retrieve from Key Vault, or a Microsoft Teams adaptive card with an expiry).
        - **Important:** The application team should retrieve these keys from a secure store, not directly from a notification.
3. **Error Handling & Rollback:**
    - What happens if the APIM subscription creation fails? The pipeline should handle this and update the work item status to "Failed" with an error message.
    - Consider implementing rollback steps or clear instructions for manual remediation.
4. **Work Item Update in Pipeline:**
    - The az boards work-item update command in the pipeline script is critical for closing the loop with the requesting team. It updates the Status and adds comments.
    - The service connection used by the pipeline must have permissions to update work items in Azure Boards.
5. **Role-Based Access Control (RBAC):**
    - The Service Principal used by your TerraformBackendVariables (or a dedicated SP for onboarding) needs:
        - API Management Service Contributor on the APIM instance.
        - Permissions to update work items in Azure Boards.

- Permissions to trigger pipelines (if using the Azure Function trigger).

This comprehensive approach allows application teams to initiate their onboarding via familiar Azure Boards, automates the provisioning with Azure Pipelines, and integrates with approval workflows to ensure governance.