ChatGPT

Application Modernization Automation Tool – Design Wiki

Overview

This design outlines a **reusable CLI tool for .NET application modernization**. The tool runs locally and leverages AI (AWS Bedrock with the Claude 3.7 model) to analyze a .NET codebase and automatically apply modernization changes. It is designed to help upgrade legacy .NET applications to newer frameworks, refactor monolithic architectures into microservices, and introduce containerization, with minimal manual effort. Key features include:

- **AI-Driven Code Analysis & Refactoring:** Uses AWS Bedrock's Claude 3.7 model to review code and suggest or implement updates (e.g. upgrading .NET Framework to .NET 6/7, replacing obsolete libraries, etc.). The model can propose code changes to take advantage of modern language features and best practices [1].
- **Iterative Modernization Loop:** Employs an iterative loop where the tool ingests the repository, lets the AI analyze the current architecture, generates a list of modernization tasks, applies one or more changes, and repeats. This continues until a stopping condition is met (max iterations, time limit, or the AI signals completion). This approach ensures gradual, safe transformation rather than one monolithic change.
- **Local Execution with External Integrations:** Runs on the developer's machine (or CI environment) and accepts a local Git repository path as input. It integrates with Azure DevOps for source control operations via a Personal Access Token (PAT), enabling automated commits/pushes to the repo. Modernized code can be pushed to a new branch in Azure DevOps for review (similar to how AWS's own .NET transform service commits changes to a new branch for safety [2]).
- **Knowledge Base Storage:** Utilizes AWS resources available in the local environment. The tool uses the developer's AWS credentials to access Amazon Bedrock and an S3 bucket. The S3 bucket acts as a **knowledge base** for the project – storing relevant information like code embeddings or serialized code content for the AI to reference across iterations. This persistent knowledge store helps handle large codebases that can't fit entirely in the model's prompt at once.
- **Configurable AI Prompts:** Ships with a library of pre-written system prompts and instructions focused on modernization themes: upgrading framework versions, refactoring to microservices, and adding containerization support. These prompts guide the AI's behavior. Developers can adjust or override these prompts via CLI options to fine-tune the transformation process or inject project-specific context.
- **Python Implementation & CLI Interface:** The entire tool is implemented in Python for accessibility and scripting flexibility. A command-line interface (CLI) is provided for configuration (e.g. specifying repo path, credentials, target modernization goals) and to trigger the automation. This makes it easy to integrate into build pipelines or run ad-hoc from a developer's terminal.

Architecture

The tool's architecture is composed of several **modular components** that work together in the modernization pipeline:

- **CLI Orchestrator (Main Module):** The entry point is a Python script (or module) that parses CLI arguments and coordinates the overall process. It loads configuration (AWS credentials, Azure DevOps PAT, prompt settings, etc.), then triggers the iterative modernization loop. This orchestrator manages state across iterations (e.g. loop count, time elapsed, tracking which tasks have been done).

- **AI Integration (AWS Bedrock Claude 3.7):** This component handles communication with the Claude model hosted on AWS Bedrock. It uses the AWS SDK for Python (boto3) to call the Bedrock **Converse** API, sending prompt messages and receiving the model's completion [3]. The Claude 3.7 model serves as the "brain" of the tool – performing code analysis, generating upgrade plans, and even producing code diffs or rewritten code segments. The Bedrock client is initialized with appropriate AWS credentials (e.g. via environment or default AWS config) and the target model ID for Claude 3.7 [4]. The model's large context window is utilized to pass code snippets or summaries for analysis.

- **Repository Ingest & Knowledge Base (S3):** There is an ingestion module that scans the input Git repository folder. It reads source files (C# code, project files, config files, etc.) and potentially **indexes or summarizes** them. The full content or summaries are stored in an Amazon S3 bucket, effectively creating a knowledge base of the application's code. Storing this data in S3 allows the AI component to fetch relevant context across iterations (for example, when the AI needs to analyze a specific module, the tool can retrieve that file's content from S3 to include in a prompt). The knowledge base might also store intermediary artifacts like the AI's analysis output or a list of identified tasks, so that each iteration can build on prior results. All S3 operations use the local AWS credentials/config – for instance, using `boto3.client('s3')` with default credentials to put and get objects.

- **Azure DevOps Integration:** To integrate with source control, an Azure DevOps module manages Git operations using the Azure DevOps REST API or via direct Git commands. The tool authenticates using a Personal Access Token (PAT) provided via CLI or environment. For example, using Microsoft's Azure DevOps Python API, one can connect to the org and project with a PAT [5] [6], allowing operations like creating a new branch, committing changes, and pushing. Alternatively, the tool can invoke Git directly (e.g. using GitPython or subprocess) and use the PAT in the remote URL for authentication. After each iteration (or after the final iteration), the tool can automatically commit the AI-generated changes to the local repo and push to Azure DevOps. By default, changes are pushed to a **new branch** (e.g. `modernization-ai-updates`) to avoid disrupting the main branch. Developers can then review a pull request with all changes.

- **Code Modification Engine:** This part of the system takes the AI's suggested code changes and applies them to the local codebase. Depending on how the AI outputs changes, this could involve parsing a diff, or simply replacing entire file contents with updated versions. The tool can back up the original files and then write the new code to disk. It ensures that formatting is preserved where possible and may run a compilation or test step to verify the changes (future extensibility). In the current design, the AI is expected to output self-contained code for replacement; the tool maps that

to the correct file path in the repo and writes it. For instance, if the AI suggests a new Dockerfile content, the tool will create or update the `Dockerfile` in the repository with the provided content.

- **Iterative Controller:** The orchestrator uses an **iterative loop** controller to drive the modernization process. In each iteration, it performs these steps in sequence:

- **Ingest & Update Knowledge Base:** Read the latest state of the repository (initially the original code, later including any changes from prior iterations) and update the knowledge base. This may include generating summaries of any new code changes to inform the next AI analysis.
- **Architecture Analysis:** Formulate an analysis prompt describing the current architecture and known legacy areas. The AI is asked to identify outdated frameworks, tightly coupled components, pain points, etc. The response is parsed to understand what needs modernization (e.g. "Project uses .NET Framework 4.x with legacy WCF services and no containerization").
- **Task Generation (Modernization Plan):** Based on the analysis, the tool prompts the AI to generate a prioritized list of modernization tasks. Each task is a actionable item such as "Upgrade project to .NET 6", "Refactor WCF service to REST API and host as separate microservice", "Add Dockerfile and containerize the application", etc. The tasks include enough detail for implementation.
- **Apply Tasks (Code Transformation):** The tool iterates through the proposed tasks (or a subset at a time) and for each one, it prompts Claude for the specific code changes required. For example, if one task is "Update the ASP.NET MVC project to ASP.NET Core", the tool might pass relevant project file content or controller code to the model with instructions to convert to ASP.NET Core patterns. The AI's response (new code or diff) is then applied to the repository. Each change is saved and optionally committed locally.
- **Review & Loop Control:** After one pass of applying tasks, the tool can either stop (if the model indicated the plan is complete or if a single iteration mode is configured) or re-analyze the updated codebase to see if further modernization is needed. If continuing, it goes back to step 1, feeding the next analysis prompt with the updated state. Loop termination criteria include:
    - A maximum number of iterations (configurable, e.g. 5 iterations) reached.
    - A time limit (e.g. stop after 1 hour of processing).
    - The AI explicitly indicating that no further tasks are needed (the plan is fully implemented).

Throughout the loop, progress and changes are logged to the console (or a log file) by the CLI orchestrator for transparency. This iterative refine-and-apply approach is inspired by agent-based automation and helps manage complex migrations step by step, rather than attempting everything in one giant leap.

Workflow Summary: The following pseudo-code illustrates how these components interact in the main orchestration flow:

```
# Pseudo-code for the main modernization loop
repo_path = args.repo
pat = args.pat
max_iter = args.max_iterations
bedrock_client = boto3.client("bedrock-runtime", region_name="us-east-1")  #
uses AWS creds
azure_conn = azure_devops_connect(org_url=args.azure_org, pat=pat)        #
Azure DevOps connection
branch_name = args.branch or "modernization-ai-updates"
```

```
# Ensure working on a new branch in the local git repo
git_checkout_new_branch(repo_path, branch_name)

finished = False
iteration = 0
while iteration < max_iter and not finished:
    iteration += 1
    print(f"=== Iteration {iteration} ===")
    # 1. Ingest repository and update knowledge base on S3
    knowledge_index = ingest_repo_to_s3(repo_path, s3_bucket=args.s3_bucket)
    # 2. AI analysis of current architecture
    analysis_prompt = prompts.get("analysis")  # predefined analysis prompt
template
    analysis_input = build_analysis_input(knowledge_index)  # assemble context
(e.g. list of components)
    analysis_response = bedrock_client.converse(
        modelId=CLAUDE_MODEL_ID,
        messages=[ {"role": "user", "content": analysis_prompt +
analysis_input} ],
        inferenceConfig={ "maxTokens": 1024, "temperature": 0 }  # using Claude
model
    )
    analysis_text = analysis_response["output"]["message"]["content"][0]["text"]
    # 3. Generate modernization tasks from analysis
    plan_prompt = prompts.get("plan")
    plan_input = f"Here is the analysis:\n{analysis_text}\nProvide a list of
modernization tasks."
    plan_response = bedrock_client.converse(
        modelId=CLAUDE_MODEL_ID,
        messages=[ {"role": "user", "content": plan_prompt + plan_input} ],
        inferenceConfig={ "maxTokens": 1024, "temperature": 0 }
    )
    tasks_text = plan_response["output"]["message"]["content"][0]["text"]
    tasks = parse_tasks_list(tasks_text)
    # Check if tasks suggest completion
    if not tasks or tasks_text.strip().lower().startswith("no further changes"):
        finished = True
        break
    # 4. Apply each task via AI guidance
    for task in tasks:
        refactor_prompt = prompts.get("refactor")  # generic refactor prompt
template
        # Optionally include relevant code snippet in prompt:
        context_code = fetch_relevant_code(task, knowledge_index, repo_path)
        user_message = f"{refactor_prompt}\nTask: {task}\n{context_code}"
        change_response = bedrock_client.converse(
            modelId=CLAUDE_MODEL_ID,
```

```
            messages=[ {"role": "user", "content": user_message} ],
            inferenceConfig={ "maxTokens": 2048, "temperature": 0 }
        )
        change_text = change_response["output"]["message"]["content"][0]["text"]
        files_modified = apply_changes_to_repo(repo_path, change_text)
        print(f"Applied task: {task} – modified files: {files_modified}")
    # Commit the changes from this iteration
    git_commit(repo_path, message=f"Iteration {iteration}: AI modernization
changes")
    # (Optionally run build/tests here to verify)
# end while

# Finalize: push changes to Azure DevOps repo (new branch)
git_push(repo_path, remote="origin", branch=branch_name, auth_token=pat)
print("Modernization complete. Changes pushed to branch:", branch_name)
```

In this pseudocode, `ingest_repo_to_s3` walks through the repository and uploads files to S3 (for knowledge base). The Bedrock `converse` calls demonstrate sending a user prompt to Claude (with appropriate model configuration) and retrieving the model's text output [3]. The actual `modelId` for Claude 3.7 would be used (e.g. `"anthropic.claude-3-7"` with the latest version suffix). Azure DevOps operations (`git_commit`, `git_push`) use the PAT for authentication (for example, by configuring the Git remote URL to include the PAT or using Azure DevOps Python API to create a push). The code ensures iterative commits so progress is saved. The loop breaks either when no more tasks are returned or when the max iteration is hit. After completion, the new branch with changes is available for review in Azure DevOps.

Setup Guide

Follow these steps to set up and configure the modernization tool in your environment:

1. **Install Python and Dependencies:** Ensure you have Python 3.8+ installed. Install the required Python packages: use pip to install AWS SDK (boto3) for Bedrock, the Azure DevOps Python API, and any other needed libraries:

   ```
   pip install boto3 azure-devops gitpython
   ```

   (The `azure-devops` package provides Azure DevOps REST clients, and `gitpython` can be used for local Git operations. You may also need other packages for parsing or prompt management as specified in the project's README.)

2. **AWS Account Setup:** You need access to Amazon Bedrock (which may be in limited preview/availability in your region) and permissions to use the Claude model. Configure your AWS credentials in one of the usual ways (environment variables, `~/.aws/credentials` file, or AWS CLI config). The tool will use the default AWS credential provider chain. Also ensure the AWS region supports Bedrock (e.g. `us-east-1` or other supported region). No explicit Bedrock API key is needed beyond

the AWS credentials. Ensure your IAM user/role has permissions for `BedrockInvoke` and S3 access. If using a specific AWS profile, set the environment variable, for example:

```
export AWS_PROFILE=myprofile
export AWS_DEFAULT_REGION=us-east-1
```

(Or you can pass a `--aws-profile` option to the CLI if implemented.)

3. **Prepare S3 Bucket:** Create or identify an S3 bucket to be used as the knowledge base store. The bucket name will be provided to the tool via configuration (e.g. `--s3-bucket my-modernize-kb`). The IAM credentials used must have read/write access to this bucket. It's recommended to use a separate bucket (or bucket prefix) for each project to avoid key collisions. No special structure is needed – the tool will create a folder-like prefix (e.g. `knowledge_base/`) within the bucket to store files.

4. **Azure DevOps Personal Access Token (PAT):** Obtain a PAT from Azure DevOps with **repository read/write scopes**. In Azure DevOps, go to User Settings > Personal Access Tokens and create a token with scopes such as *Code (Read & Write)* for the relevant project or organization. Note down this token (a long string) securely. You will provide it to the tool either as a CLI argument (`--pat`) or through an environment variable. The PAT is used for authenticating Azure DevOps API calls or git pushes. For example, the tool can establish a connection like:

```python
from azure.devops.connection import Connection
from msrest.authentication import BasicAuthentication

personal_access_token = "YOUR_PAT_HERE"
organization_url = "https://dev.azure.com/YourOrg"

credentials = BasicAuthentication('', personal_access_token)
connection = Connection(base_url=organization_url, creds=credentials)
git_client = connection.clients.get_git_client()
```

This uses Azure DevOps's Python SDK to authenticate with the PAT and get a Git client [5] [6]. The tool can then call methods to create a new branch or push commits via the Azure DevOps REST API. (If the tool uses direct Git operations instead, it would configure the git remote like `https://user:$PAT@dev.azure.com/Org/Project/_git/Repo` for pushing.)

1. **Local Repository Ready:** Ensure you have the .NET application's code in a local Git repository. You can clone the Azure DevOps repo to your machine or use an existing local copy. Provide the path to this repository when running the tool. Make sure the repo is **clean** (no uncommitted changes), so the tool can manage commits. It's also recommended to be on a new branch or to allow the tool to create one, so as not to override any important work on your main branch.

2. **Configuration of Prompts (Optional):** The tool comes with default prompt templates for the AI. If you wish to customize the prompts (for example, to insert your company's coding guidelines or specify a target .NET version), prepare a prompt configuration file (JSON or YAML as required by the tool). You can then pass this file path via CLI (e.g. `--prompt-config myprompts.json`). If you don't have custom prompts, the system defaults will be used.

3. **Run a Dry Test (Optional):** It might be wise to run the tool in a non-destructive mode initially (if supported), such as a "planning" mode where it only outputs the suggested plan without applying changes. If the tool has a flag like `--dry-run` or `--plan-only`, use that to see what it *would* do. If not, you can run with a low iteration count (e.g. `--max-iter 1`) to just get an analysis and initial tasks, and review them manually.

4. **Running the Tool:** Invoke the CLI with required arguments. For example:

   ```
   $ python modernizer.py \
       --repo "/path/to/MyLegacyApp" \
       --azure-org "https://dev.azure.com/MyOrg" \
       --pat "azd_pat_123...XYZ" \
       --s3-bucket "my-modernize-kb" \
       --branch "modernize-upgrade-dotnet6" \
       --max-iter 3 \
       --time-limit 1800  \
       --prompt-config "./custom_prompts.json"
   ```

   This command points the tool at the local repo, Azure DevOps org, PAT, and S3 bucket. It also sets a branch name for output, limits to 3 iterations or 30 minutes (1800 seconds), and uses a custom prompt configuration file. Adjust these parameters as needed. If no branch name is given, a default (like `modernization-ai-updates`) is used. Omit `--prompt-config` to use defaults. Once run, the tool will begin analyzing and modernizing the app per the configured options. Progress will be logged to the console.

5. **Post-run Review:** After the tool finishes, go to your Azure DevOps project. You should find a new branch (as specified) with the committed changes. Review the commit history and diff. The tool may also output a summary of changes or a modernization report in the console or as a file. It's crucial to **review and test** the changes. Ensure the app builds and passes tests. AI changes might need tweaking – the tool aims to save effort but isn't guaranteed to produce perfect results [1]. Run your test suite or manually test the modernized application. You can then merge the branch via a Pull Request after approval.

By following this setup guide, you prepare the environment for the tool to function correctly. Most of the heavy lifting (AI calls, file manipulations, etc.) is handled by the tool once configured.

CLI Usage

The tool is operated through a command-line interface. Below is the general usage and available commands/flags:

```
Usage: modernizer.py --repo REPO_PATH --azure-org ORG_URL --pat PAT_TOKEN --s3-
bucket S3_NAME [options]

Options:
  --repo REPO_PATH        Path to the local Git repository of the .NET app to
modernize (required).
  --azure-org ORG_URL     Azure DevOps organization URL (e.g. https://
dev.azure.com/MyOrgName).
  --pat PAT_TOKEN         Azure DevOps Personal Access Token with repo access
(required).
  --s3-bucket S3_NAME     Name of the AWS S3 bucket for knowledge base storage
(required).
  --branch BRANCH_NAME    Name of the Git branch to create/use for committing
changes (default: modernization-ai-updates).
  --max-iter N            Maximum number of iterations for the modernization
loop (default: 5).
  --time-limit SECONDS    Maximum time (in seconds) to run the tool before
stopping (default: no limit).
  --prompt-config FILE    Path to a JSON/YAML file containing custom prompt
overrides (default: use built-in prompts).
  --target-framework VER  Target .NET version or framework to upgrade to (e.g.
net6.0, net7.0). Used to guide prompts.
  --no-apply              If set, the tool will only analyze and plan, but not
apply changes (dry run mode).
  --help                  Show this help message and exit.
```

Basic Example: To run the tool on a repository, you might use:

```
modernizer.py --repo "./LegacyApp" --azure-org "https://dev.azure.com/Org" \
  --pat "xxxx" --s3-bucket "my-kb-bucket" --target-framework "net6.0"
```

This would analyze the `LegacyApp` repo and attempt to modernize it to .NET 6.0, using default prompts and settings, and then push changes to a new branch on Azure DevOps.

Explanation of Key Arguments:

- `--repo` **(required):** Filesystem path to the local Git repository containing the application code. The tool will operate on this folder (reading files, writing changes, and performing Git commits). The repo should be a Git repo with a remote on Azure DevOps (so that pushing via PAT is possible). If the

repo is not already cloned, you must clone it first or allow the tool to clone (not implemented by default, so usually you clone manually).

- `--azure-org` **(required):** The base URL of your Azure DevOps organization or collection. For example, `https://dev.azure.com/MyOrg` or the older format `https://MyOrg.visualstudio.com`. This is used in API calls and to construct Git push URLs. The tool uses this along with the PAT to authenticate. (Project and repo names are inferred from the local repo's Git remote configuration, or additional flags could be provided to specify them if needed.)

- `--pat` **(required):** The Personal Access Token for Azure DevOps authentication. It's recommended not to put this in plain text on the command line for security reasons (as it could be visible in process lists or shell history). Instead, you can omit this flag and the tool will prompt securely for the token, or you can set an environment variable that the tool picks up. For example, `AZURE_DEVOPS_PAT=xxxxx modernizer.py --repo ...` if the tool is coded to read from `AZURE_DEVOPS_PAT`. The PAT is used to authorize Git operations (push commits to remote) and could also be used if the tool calls Azure DevOps REST APIs directly (e.g. to create a PR automatically).

- `--s3-bucket` **(required):** The name of the AWS S3 bucket used for storing knowledge base data. This must be in the same AWS account/region configured in your AWS credentials. The tool will create objects in this bucket to store code context (and possibly remove them when done). Make sure the bucket exists **before** running (the tool doesn't auto-create buckets for safety). You can include a prefix as part of the name if needed (e.g. `mybucket/knowledge/project1`), or the tool may use a fixed prefix internally.

- `--branch`**:** The Git branch name to use for commits. If not specified, a default (like `modernization-ai-updates`) will be created. If the branch already exists locally, the tool will attempt to use it (fast-forwarding if possible). All AI-driven changes will be committed to this branch. This allows you to isolate AI changes from other work and review them. After the run, you'd typically create a PR from this branch to your main branch. If you want the tool to commit directly to an existing branch (not generally recommended), you can specify that branch name (ensure you have it checked out or the tool will create it if missing).

- `--max-iter`**:** Maximum iterations of the analyze->plan->apply loop. By default it might be 5, meaning it will do up to 5 rounds of analysis and code modifications. If your application is very large or very legacy, you might increase this. Conversely, you can set it to 1 to do a single-pass modernization (e.g. just get one set of changes). The tool will stop early if it finishes all tasks in fewer iterations.

- `--time-limit`**:** A safety cutoff in seconds. This ensures the tool doesn't run indefinitely. For instance, `--time-limit 3600` would stop the process after an hour, even if iterations are still remaining. The current iteration will finish its task application before stopping. If not set, the tool will run until completion or max iterations. This is useful in CI pipelines to avoid timeouts or if the Bedrock API calls are slow and you have a limited time window.

- `--prompt-config`: Path to a file with custom prompts. This allows overriding the default prompt templates. The file could be JSON or YAML containing the same keys that the tool expects (e.g. "analysisPrompt", "planPrompt", "refactorPrompt", etc.). For example, you might have a JSON like:

  ```
  {
    "analysisPrompt": "You are a .NET modernization expert. Analyze the following code base for legacy patterns...",
    "planPrompt": "Given the analysis, list the steps to modernize the application to .NET 6 and containerize it.",
    "refactorPrompt": "Perform the following code update task. Respond with the changed code only..."
  }
  ```

  If provided, the tool will merge these with its defaults, so you can override only certain aspects. If not provided, the built-in prompts tailored for .NET modernization (as described in the next section) will be used.

- `--target-framework`: A convenience option to specify the desired target .NET version or platform. For example, `--target-framework net7.0` might instruct the AI to specifically target .NET 7. This flag can influence the prompt content (the tool might insert "target version: .NET 7" in prompts) or provide context to the AI so that the generated code uses correct APIs for that version. If omitted, the AI will choose a reasonable modern target (often the latest LTS .NET). This is useful if you have a mandate to go to a specific version or to Azure cloud services, etc.

- `--no-apply`: If this flag is present, the tool will skip the actual application of changes. It will still run analysis and planning, and even generate the change diffs, but it will not write them to the repository or commit. Instead, it might output the suggested changes to the console or a file. This is essentially a **dry run** mode. Use this if you want to see what the AI *would* change before letting it modify files, or if you intend to apply changes manually. It's also helpful for getting an architectural analysis and task list without committing to changes. By default, if `--no-apply` is not set, the tool will apply and commit changes automatically.

After running the CLI, you will see log messages indicating progress. For example, the tool will print which iteration is running, what tasks were identified, and which files have been modified. If verbose logging is enabled (you might add a `--verbose` flag in the future), it could also print the AI's raw output for inspection. On errors (like failing to call Bedrock or if Git commit fails), the tool will raise an error and stop. Use the `--help` flag to see the latest usage instructions as the tool's interface may evolve.

Here is a snippet of how the CLI parsing might be implemented in Python using `argparse`:

```
import argparse

parser = argparse.ArgumentParser(description="Automate modernization of .NET applications using AI")
parser.add_argument('--repo', required=True, help='Path to local Git repository
```

```
of the app')
parser.add_argument('--azure-org', required=True, help='Azure DevOps
organization URL (https://dev.azure.com/Org)')
parser.add_argument('--pat', required=True, help='Azure DevOps Personal Access
Token (Repo RW scopes)')
parser.add_argument('--s3-bucket', required=True, help='S3 bucket name for
knowledge base storage')
parser.add_argument('--branch', default='modernization-ai-updates', help='Git
branch for committing changes')
parser.add_argument('--max-iter', type=int, default=5, help='Max iterations for
the AI modernization loop')
parser.add_argument('--time-limit', type=int, help='Time limit in seconds for
the tool to run')
parser.add_argument('--prompt-config', help='Path to custom prompt definitions
(JSON/YAML)')
parser.add_argument('--target-framework', help='Target .NET version (e.g.,
net6.0, net8.0)')
parser.add_argument('--no-apply', action='store_true', help="Analyze and plan
only, do not apply changes")
args = parser.parse_args()
```

This sets up the command-line interface. Once `args` are parsed, the main program would proceed to use those arguments (for example, establishing the Azure DevOps connection with `args.pat` and `args.azure_org`, initializing AWS Bedrock client, etc.). The design of the CLI emphasizes required inputs for anything that the tool cannot default (like the repo path and credentials), while optional flags allow controlling the process.

Prompts Library

One of the core strengths of this tool is the curated **prompts library** that guides the AI model (Claude 3.7 via Bedrock) through the modernization workflow. These prompts are essentially instructions or questions given to the AI at various stages: analyzing the codebase, creating a modernization plan, and executing specific refactoring tasks. They are designed to be *detailed and context-rich*, since the quality of prompts directly influences the usefulness of the AI's output.

Default Prompt Templates: The tool includes a set of default prompt templates covering different purposes. They are written to reflect best practices in .NET modernization. Below are the key prompts and their roles:

- **System Prompt (Role Instructions):** This is a prompt given to Claude to establish its role and tone. For example: *"You are an expert .NET software architect and refactoring assistant. You have extensive knowledge of all .NET versions, cloud-native architecture, microservices design, and containerization. You will help modernize a legacy codebase. Provide detailed and precise recommendations and code changes. If you show code, make sure it's complete and valid."* This system-level instruction ensures the model acts as a specialized assistant for our domain.

- **Architecture Analysis Prompt:** Used in the first step of each iteration, asking the model to analyze the entire application. It might look like:
  Prompt Name: `analysisPrompt`
  Content (template):

```
Analyze the current architecture and implementation of the .NET application
provided. Identify:
1. Legacy frameworks or outdated technologies (e.g., .NET Framework
version, old libraries like Web Forms or WCF).
2. Monolithic areas that could be broken into services.
3. Opportunities to introduce cloud-native or modern practices (like
dependency injection, configuration management improvements).
4. Any issues with the code that impede modernization (tight coupling, lack
of abstraction, etc.).
Provide a summary of the application's architecture and the main points
that need modernization.
```

  The actual implementation will append or include a representation of the codebase. Since the entire code won't fit in the prompt, the tool might include a high-level summary: e.g., a list of projects, frameworks detected, major components, and possibly snippet of important files like the startup configuration. The AI's response to this prompt is expected to be a **descriptive analysis** of the system (e.g., "This is a .NET Framework 4.7.2 web app using ASP.NET MVC, with a tightly coupled architecture and direct database calls. It lacks an API layer, and uses outdated packages..."). That gives context for the next step.

- **Modernization Plan Prompt:** After analysis, we ask Claude to formulate a plan.
  Prompt Name: `planPrompt`
  Content:

```
Based on the analysis of the application, list the recommended
modernization tasks to bring the app up to date.
Each task should be specific and actionable, mentioning what to change and
the goal. Focus on:
- Upgrading to the latest .NET (or the target framework specified).
- Refactoring or redesigning for a microservices or modular architecture if
applicable.
- Containerizing the application (e.g., adding Docker support).
- Improving any other aspects (performance, security, DevOps) relevant to
modernization.

Format your answer as an ordered list of tasks. For each task, provide a
short rationale if needed.
```

  With this prompt, the AI will produce something like a numbered list:

- **Upgrade .NET Framework to .NET 6** – Convert the projects to SDK-style and update NuGet dependencies to their .NET 6 equivalents.
- **Refactor WCF Services to REST API** – Replace legacy SOAP services with ASP.NET Core Web API endpoints, separating them into a new service project.
- **Introduce Docker Containers** – Write Dockerfile(s) for the application and database, containerize the app for deployment on Kubernetes or ECS.
- **Decouple DAL from UI** – Introduce a repository pattern or microservice for data access to prepare for future scaling... And so on. These tasks guide what changes will be made.

- **Refactoring/Task Execution Prompt:** This prompt (or set of prompts) is used for each specific task to generate code changes. We often include relevant code context in the prompt to help the model.
  Prompt Name: `refactorPrompt` (generic, or there might be specialized ones per task type)
  Content Template:

````
Task: {task_description}
The following code is an excerpt related to this task:
```csharp
{code_snippet}
````

Refactor/update the code above to accomplish the task. Provide the *full updated code* for the snippet or file, incorporating any necessary changes.

- If upgrading framework or libraries, adjust project file or using statements accordingly.
- If refactoring architecture, modify the design (e.g., show new classes, updated methods).
- If adding containerization, you may provide a Dockerfile content or configuration changes. Your response should contain only the updated code or the specific changes (in context), without additional explanation.

```
The `{task_description}` is replaced with something like “Upgrade the project’s .csproj to .NET 6” or “Add a Dockerfile to containerize the web app.” The `{code_snippet}` is fetched by the tool, e.g., if the task is about upgrading project file, the snippet might be the old `.csproj` content; if about adding Dockerfile, maybe nothing exists yet, so we ask for new content outright. The instructions tell the AI to focus on outputting code. We might use separate prompts for different scenarios:
  - *Upgrade Prompt:* specifically for upgrading project files or package references.
  - *Microservice Prompt:* guiding the model to extract part of code into a new service (this is complex; the model might suggest a design and code for the new service).
  - *Docker Prompt:* to generate Dockerfile or Docker Compose snippets.
  - *Code Fix Prompt:* for simpler transformations (like using newer language syntax or fixing security issues).
```
```

````
In practice, the tool could have a library like:
```python
default_prompts = {
    "analysis": ANALYSIS_PROMPT_TEXT,
    "plan": PLAN_PROMPT_TEXT,
    "refactor": REFACTOR_PROMPT_TEXT,
    "docker": DOCKER_PROMPT_TEXT,
    ...
}
````

and selects which prompt to use based on the task. For simplicity, one generic `refactorPrompt` might be used with the task description embedded.

- **Containerization Prompt (specific example):** If a task is about containerization, the tool could use a specialized prompt to generate a Dockerfile. For example:

  ```
  The application needs to be containerized. Write a Dockerfile for a .NET
  application based on {target_framework}.
  Include steps to build the application (if needed) and to run it. Use best
  practices for a production-ready container (e.g., using ASP.NET base
  images, multi-stage builds).
  Also suggest any configuration changes required for the app to run in a
  container (like environment variable usage).
  ```

  The AI would then output a Dockerfile content. The tool would save this as `Dockerfile` in the repo. Similarly, if Kubernetes deployment is in scope, a prompt could ask for a sample Kubernetes manifest, though that's beyond initial scope.

All these prompts are **adjustable**. The developer using the tool can supply a `--prompt-config` file to override parts of these templates. For instance, you might want the plan to always include a step about adding unit tests, so you could append that to the plan prompt template. Or you might be targeting Azure PaaS deployment instead of containers, so you could change the containerization prompt to something about Azure App Service configuration.

Prompt Tuning and Context: The prompts will often be combined with context data. Because .NET projects can be large, the tool must judiciously select what information to feed the model. Strategies include: - Providing high-level summaries of each project (e.g., "Project A: ASP.NET MVC on .NET 4.7, uses EntityFramework 6, etc."). - Feeding representative code files (like an example controller, a data access class, etc.) to illustrate patterns. - When focusing on a specific task, provide only the files relevant to that task (e.g., for upgrading a project file, just provide the project file content). - Possibly using the Claude model's ability to handle very large inputs (Claude 3.7 is known for large context windows, potentially 100k tokens) [7], meaning the tool could even include a lot of code text if needed. However, to be safe and efficient, summarization and step-by-step feeding is preferred.
  ```

Example of Combining Prompt and Context in Code:
Here's how the tool might prepare a prompt for a specific task in code form:

```
task = "Refactor the legacy WCF service to a RESTful Web API using ASP.NET
Core."
# Suppose we identified Service.cs as the WCF service implementation:
service_code = open("LegacyApp/Service.cs").read()
prompt_template = prompts["refactor"]  # the template text as discussed
user_prompt = prompt_template.format(task_description=task,
code_snippet=service_code)
response = bedrock_client.converse(
    modelId=CLAUDE_MODEL_ID,
    messages=[{"role": "user", "content": user_prompt}],
    inferenceConfig={"maxTokens": 1500, "temperature": 0.2}
)
updated_code = response["output"]["message"]["content"][0]["text"]
```

In this snippet, we inject the actual code (`service_code`) into the prompt under the code fences. The model then returns `updated_code` which should contain the refactored service (e.g., a controller or a minimal API). The low temperature (0.2) is used to reduce randomness, aiming for a deterministic refactor following instructions.

The **prompts library** is central to the tool's flexibility. By editing the prompts, one could adapt the tool to *different languages or frameworks* as well. For example, to modernize Java apps, one would adjust these templates to talk about Java, Spring Boot, etc. The structure of analysis-plan-execute would remain, just the content of prompts changes.

Finally, note that prompt engineering is iterative. If the AI's output isn't as expected, the prompts may be refined. The system prompt can also enforce certain style guidelines (e.g., "only output valid C# code, no explanatory text"). The provided defaults are a starting point, and we expect users (or future versions) to improve them as the tool is used on real projects.

Dev Guide

This section is for developers who want to understand or extend the internals of the tool. We'll discuss the code structure, key components, and how they work together. The tool is written in Python and organized for clarity:

Project Structure:

```
modernizer/
├── modernizer.py         # Main CLI orchestrator
├── prompts.py            # Default prompt templates (as constants or loaded
from a file)
├── ai_client.py          # AWS Bedrock client wrapper
```

```
├── repo_ingest.py        # Functions to ingest code and manage S3 knowledge
base
├── azure_devops.py       # Functions for Azure DevOps integration (Git
operations, PR creation)
├── apply_changes.py      # Logic for applying AI-proposed changes to the
filesystem
└── utils.py              # Utility functions (e.g., parsing AI output, diff
handling, logging)
```

(This is a conceptual layout; the actual implementation may group things differently or use classes to encapsulate state.)

Main Orchestrator (`modernizer.py`):
This module uses `argparse` to parse CLI arguments (as shown earlier). After parsing, it likely performs: - AWS and Azure setup: e.g., load AWS creds (via boto3 which automatically picks up env vars or profile) and instantiate an Azure DevOps connection or configure Git. For Azure, you might use the `azure_devops` module or simply set up Git credentials. For example, if using GitPython, you might do:

```
import git
repo = git.Repo(args.repo)
if args.branch not in repo.heads:
    repo.git.checkout('-b', args.branch)
else:
    repo.git.checkout(args.branch)
# Set remote URL with PAT for push:
origin = repo.remote(name='origin')
pat_url = args.azure_org.replace('https://', f'https://:{args.pat}@')
origin.set_url(pat_url + repo.remote().url.split(args.azure_org)[-1])
```

This is one way to configure the PAT for pushing (embedding the PAT in the URL, using an empty username – the colon before PAT is intentional). Alternatively, use the Azure DevOps client to push commits via API.

- Load prompts: If a custom prompt file is provided, load it (JSON/YAML) and override the defaults from `prompts.py`. This results in a dictionary of prompt strings to use.

- Initialize the Bedrock client:

```
import boto3
bedrock = boto3.client('bedrock-runtime', region_name=args.aws_region or
"us-east-1")
CLAUDE_MODEL_ID = "anthropic.claude-3-7-...."  # the exact model ID from
AWS (could also be a config)
```

The model ID might be something like `anthropic.claude-3-7-2025-...:0` as per AWS documentation [8]. Using boto3 with Bedrock requires AWS to allow Bedrock usage in that account. The client will use the AWS creds configured (you don't manually pass keys if environment is set up).

- Enter the iterative loop (as shown in pseudocode earlier). Likely this is implemented in a function that can be called recursively or iteratively. It will call out to helper functions:

  - `ingest_repository()` from `repo_ingest.py` to upload files to S3 or at least generate a summary of code. This function might also return a data structure (like a dict of file paths to S3 keys or to local content) that can be used for context. For example:

    ```
    def ingest_repository(repo_path, bucket):
        import os, boto3
        s3 = boto3.client('s3')
        knowledge_index = {}
        for root, _, files in os.walk(repo_path):
            for fname in files:
                if fname.endswith((".cs", ".config", ".csproj")):  # 
    focus on relevant files
                    full_path = os.path.join(root, fname)
                    rel_path = os.path.relpath(full_path, repo_path)
                    content = open(full_path, 'r', 
    encoding='utf-8').read()
                    key = f"knowledge_base/{rel_path.replace(os.sep, 
    '/')}"
                    s3.put_object(Bucket=bucket, Key=key, Body=content)
                    knowledge_index[rel_path] = key
        return knowledge_index
    ```

    This simplified example uploads C# source files and project files to S3, and builds an index mapping relative file paths to S3 object keys. In practice, you might filter out large binary files or dependencies. The knowledge_index can be used later to fetch content when needed (`s3.get_object`). If the repository is huge, an alternative is to not upload everything, but perhaps only store summaries; however, storing raw content ensures nothing is lost. The S3 storage is also useful if the AI agent were to have retrieval abilities or if you want to persist the state between runs.

  - `analyze_architecture()` in `ai_client.py` might construct the analysis prompt and call Bedrock. It could also chunk the analysis if needed (e.g., first ask high-level questions then dive deeper). But a single call as in the pseudocode often suffices if prompt is well-formed.

  - `generate_plan()` in `ai_client.py` takes the analysis text and asks Claude for a task list (the planning prompt).
  - `parse_tasks_list()` in `utils.py` would parse the AI's response into a Python list of tasks (splitting by line numbers, etc.).

- Then for each task, either the orchestrator itself or a helper like `apply_task(task, knowledge_index)` handles generating and applying changes. This likely uses:
- `ai_client.py -> get_changes_for_task(task, knowledge_index)`: Prepares the refactoring prompt, fetches relevant code context (maybe using the knowledge index to get file content from S3 or directly from disk), calls Claude, and returns the new code. Possibly returns a diff or full file text.
- `apply_changes.py -> apply_change(diff_or_code)`: If the AI returned a diff (unified diff format or just code snippet), this module applies it. If full code, it might overwrite the file or insert code. This can be complex; for now assume the AI returns the full updated file content or a well-defined patch we can apply with `difflib` or similar.
- Commit and loop logic is handled in orchestrator: after each task or each iteration, using GitPython or subprocess calls to `git`:

```
repo.git.add(update=True)  # stage all changes
repo.git.commit(m=f"Apply modernization task: {task}")
```

or if batching per iteration, commit once after all tasks in that iteration. By iteration end, push is either done or deferred to the very end.

- Loop control checks the time elapsed (track start time and break if now - start > time_limit) and iteration count.

- End of processing: once done, if `--no-apply` was not set, the orchestrator triggers a push to remote. If using GitPython:

```
origin.push(refspec=f"{args.branch}:{args.branch}")
```

with authentication handled by the earlier remote URL set or using Git credential helper. Alternatively, use Azure DevOps API:

```
git_client.create_pull_request(...)
```

to perhaps open a PR. For now, pushing the branch is sufficient.

Azure DevOps Integration Considerations:
Instead of direct git, one could use the Azure DevOps REST API for commits. The Azure DevOps Python API (via `git_client`) has methods like `get_repository`, `get_refs`, `create_push`. For example, you can construct a Push object with commits. However, that is more involved than using Git locally. The current design favors using the local Git repo and then pushing via normal Git over HTTPS (with PAT as auth). This is simpler and leverages Git's handling of merges, conflicts, etc. It does require that the PAT has permission and that the remote URL trick works. Alternatively, one could configure a Git credential helper that uses the PAT.

Parsing AI Outputs:
In development, you must handle the variety in AI outputs. Sometimes the model might include explanations or not format the answer exactly as hoped. We mitigate this with prompt instructions ("provide only the code"), but it's not foolproof. The `utils.py` might include functions to clean the AI output. For example, remove any leading comments or code fences from the response. If the model returns a diff format (some advanced prompt might ask for unified diff), then `apply_changes.py` would parse that diff and patch the files. Using Python's `difflib` or simply shelling out to `patch` could apply a unified diff to the repo.

Error Handling & Logging:
From a dev perspective, important places to add error handling: - AWS Bedrock API calls: use try/except around `client.converse`. Bedrock could return errors if the prompt is too long, or if model service is unavailable. We should catch exceptions (`botocore.exceptions.ClientError`) and either retry or abort with a clear message [9]. - Git operations: if a merge conflict or commit issue arises, catch those and log. In an automated setting, we might choose to abort all changes if a conflict arises that AI can't handle. - Azure DevOps API: if using it, handle HTTP errors (unauthorized, etc.) and print meaningful messages (e.g., "PAT is invalid or lacks permissions").

During development, using verbose logging (maybe Python's `logging` library) is useful. One might include a debug mode where all AI prompts and responses are saved to files for later analysis.

Testing the Tool:
As a developer, to verify the tool: - Unit test the prompt generation functions (given a known input, do they produce the expected prompt string?). - Perhaps stub the Bedrock client during tests (since we can't call the real AI in unit tests). You might create a fake `ai_client` that returns predefined outputs for known prompts, to test the loop logic. - Integration testing can be done on a small dummy .NET project (include a sample legacy project in the repo for this purpose). Run the tool on it and see if it produces the intended output.

Performance considerations:
The loop can be time-consuming mainly due to AI calls. We set the model's `maxTokens` parameter to control output length [10] and possibly limit temperature for deterministic output. Each converse call could take a few seconds to over a minute depending on input size. If the codebase is very large, summarization might be needed to keep prompts within limits. As a dev, consider implementing a summarizer (maybe using a smaller model or heuristics) to pre-process code into shorter descriptions for the analysis prompt.

Extensibility for Devs:
The code is organized to allow swapping components (see next section). For example, all Bedrock interactions are in `ai_client.py` behind a simple interface. If one wanted to use OpenAI API instead, they could write a new client and plug it in without changing the rest of the pipeline. Similarly, the repository interface (Git/Azure DevOps) could be abstracted to allow other version control systems.

In summary, the Dev Guide covers how the tool works under the hood, providing a blueprint for maintenance and enhancement. With this knowledge, contributors can confidently extend functionality or fix issues, such as improving how diffs are applied or optimizing the prompt content for better AI performance.

Extensibility

One of the goals of this modernization automation tool is to be **extensible and adaptable** to different scenarios. While it currently focuses on .NET and uses specific technologies (AWS Bedrock, Azure DevOps), it's built in a way that key pieces can be modified or replaced without reworking the entire system. Here are some ways the tool can be extended:

- **Support for Other AI Models or Providers:** The design abstracts the AI interaction in a client module. Today it uses AWS Bedrock with Claude 3.7, but you could extend it to other models. For example, if you want to use OpenAI's GPT-4 or an open-source model like CodeLLM, you can implement a new `ai_client` that calls that API. As long as it exposes similar methods (e.g. `analyze_architecture()`, `generate_plan()`, `get_changes_for_task()`), the orchestrator can use it. This might involve changes in prompt style (different models have different strengths), but the high-level logic stays the same. You could even make the AI backend selectable via CLI (e.g., `--ai-backend bedrock|openai|local`). Each backend would have its config (API keys, endpoints). This flexibility ensures the tool isn't locked into a single vendor.

- **Multi-Language Support:** While currently tailored for .NET (C#) modernization, the framework of analysis and iterative refactoring can apply to other languages (Java, Python, etc.). To extend to another language, you'd primarily develop a new set of prompt templates and possibly some language-specific parsing. For instance, to modernize a Java monolith to Spring Boot microservices, you'd adjust the prompts to talk about Java, use a Java-savvy model (maybe CodeWhisperer or GPT-4 with Java context), and change file filters in `repo_ingest` (e.g., include `.java` files). The knowledge base storage and loop logic wouldn't need fundamental changes. This modularity means an organization could use the same tool for a variety of legacy modernization efforts, not just .NET.

- **Extending Modernization Scenarios:** The default tasks focus on framework upgrade, microservices, and containerization. You can extend the **Prompts Library** to include other modernization aspects:

- **UI Modernization:** e.g., migrating from ASP.NET Razor pages to Blazor or from WinForms to web. Add prompts to guide UI refactor (though that's a big task for AI alone).
- **Cloud Services Migration:** e.g., moving from on-prem SQL to AWS RDS or Azure SQL, or from file system to S3. Prompts could be added to detect local dependencies and suggest cloud alternatives.
- **Security Hardening:** prompts to do things like add OAuth authentication, or implement safe coding practices (some of this overlaps with static analysis).
- **Performance Tuning:** e.g., identify inefficient code and propose using async/await or better algorithms.
- **Adding CI/CD Config:** e.g., generating a GitHub Actions or Azure Pipeline YAML for the modernized app, Docker image build, etc.

Each new scenario might require new tasks and prompt templates. The tool could allow plugin modules or simply additional cases in its plan generation. For example, if a config flag `--include-tests` is set, it might append a task "Add unit tests for critical components" and have the AI generate some test stubs.

- **Interactive Mode:** An extension could be an interactive CLI or even a simple GUI where the developer approves tasks or changes before they're applied. In the current automation, it's fully automated, but a semi-automated mode could increase trust. This might involve pausing after each

task, showing a diff, and asking the user to confirm (or edit) before proceeding. Implementing this would involve integrating a diff viewer or just printing to console and waiting for input. It's an extension to improve usability for those who want more control.

- **Integration with Build/CI Systems:** The tool could be packaged as a CLI for local use and also integrated into CI pipelines. For example, you could have a Jenkins or GitHub Actions job that runs the tool on a branch and pushes the changes, then maybe runs tests. For better integration, one might add features like:

- Exiting with a non-zero code if the modernization introduces build failures (i.e., incorporate a build/test step and check results).
- Posting comments on the Pull Request summarizing what was done.
- Splitting commits per task or grouping logically, for easier review (the tool could commit each task separately rather than one big commit).

These enhancements make it friendlier for team adoption. They can be added by writing some additional scripting around the tool or enhancing the Azure DevOps integration (e.g., use the DevOps API to create a PR automatically, with a description of changes).

- **Alternate Knowledge Base Options:** Currently, the tool uses S3 to store code context. This is simple and works, but one could integrate a more advanced knowledge base:
- **Vector Databases:** Index code segments by embedding vectors (perhaps using an embedding model) to enable semantic search for relevant context. An extension could add a step where each code file is vectorized (using an AWS SageMaker endpoint or local model), stored in a vector store (like Pinecone or even an open-source library), and then for a given task, the tool queries the store for the top-N relevant code pieces to feed into the prompt. This is akin to retrieval augmented generation.

- **Local Disk as KB:** If one doesn't want to use S3, the tool could be configured to keep the knowledge base locally (just reading files directly or storing summaries in a local cache). Abstracting the storage layer would allow this. Right now, `repo_ingest.py` could be adjusted to either upload to S3 or just keep a Python object mapping file paths to content. S3 was chosen for potential distributed scenarios or if the model could directly fetch (in future AWS might allow model to read from S3), but it's not strictly necessary for functionality.

- **Logging and Monitoring:** For enterprise use, one might integrate with monitoring tools. For example, log each AI call's prompt and response to CloudWatch or Azure App Insights for auditing. Or measure how long each step takes and log metrics. Adding these doesn't change core logic but is an extension for production readiness. A developer could add a logging decorator around the AI calls to capture prompt/response pairs (with sensitive info removed) for debugging model behavior.

- **Updating for New Claude Versions or Models:** As AWS Bedrock updates models (Claude 4, etc.), the tool can be extended to support new model IDs or capabilities (like if a future model supports direct code diff generation). This might simply be a configuration change (new model_id, updated prompt phrasing) to leverage improvements.

The design's modular approach ensures that adapting the tool in the above ways is straightforward. For example, to add Java support, you mostly add new prompts and maybe a language option; to use OpenAI, you implement a new AI client with the same interface and select it via a flag. The core loop and process remain valid.

Example – Adding a New Prompt for Unit Tests: Suppose you want the tool to also add unit tests for critical functions as part of modernization. You'd do the following: 1. Create a new prompt template, e.g., `testPrompt`: "Generate a unit test class for the following class/method. Use xUnit and Moq as needed. Focus on [functionality]. Provide the complete test code." 2. Modify the plan generation to include a task like "Add unit tests for module X" (the AI might add it itself, but you can force it by template or by injecting tasks). 3. Extend the loop: after applying main tasks, or as an additional iteration, call Claude with `testPrompt` for selected classes. 4. Apply the resulting test code by writing new test files. 5. Commit those as well.

This new feature doesn't break existing ones; it's additive. The tool's flow can accommodate it either as another step in each iteration or a special final phase.

In summary, the modernization automation tool is designed not as a one-off script, but as a foundation that can evolve. By tweaking prompts, swapping out AI or storage components, and adding new capabilities, it can remain useful as technologies change (new .NET versions, new AI models, etc.). The emphasis on configuration and modular design helps ensure that developers can extend it to fit their unique modernization needs while reusing the robust framework that orchestrates the complex process of automated code transformation.

Sources:

- AWS Bedrock and Claude integration (AWS SDK for Python example) [4] [3]
- Azure DevOps PAT authentication via Python API [5] [6]
- AWS .NET modernization service committing changes to new branch [2]
- Commentary on AI-assisted .NET upgrades (caution to review AI changes) [1]

---

[1] [7] How I'd use generative AI to modernize an app – Richard Seroter's Architecture Musings
https://seroter.com/2024/02/21/how-id-use-generative-ai-to-modernize-an-app/

[2] AWS Transform for .NET, the first agentic AI service for modernizing .NET applications at scale | AWS News Blog
https://aws.amazon.com/blogs/aws/aws-transform-for-net-the-first-agentic-ai-service-for-modernizing-net-applications-at-scale/

[3] [4] [9] [10] Invoke Anthropic Claude on Amazon Bedrock using Bedrock's Converse API - Amazon Bedrock
https://docs.aws.amazon.com/bedrock/latest/userguide/bedrock-runtime_example_bedrock-runtime_Converse_AnthropicClaude_section.html

[5] [6] GitHub - microsoft/azure-devops-python-api: Azure DevOps Python API
https://github.com/microsoft/azure-devops-python-api

8 Use Anthropic Claude 3.7 Sonnet's reasoning capability on Amazon Bedrock - Amazon Bedrock
https://docs.aws.amazon.com/bedrock/latest/userguide/bedrock-runtime_example_bedrock-runtime_Converse_AnthropicClaudeReasoning_section.html