ChatGPT

AIUOKEEP Implementation Files

Below is a complete set of implementation-ready files covering all phases of the AI-operated pipeline as specified by the **AIUOKEEP** manifest [1] [2]. We group files by directory, following the manifest from **AIUOKEEP.md** [1] and using proven modular patterns [3] [4]. Each script and config is written in PowerShell or Python (3.12), uses strong typing and clear contracts, and includes comments and tests per the 5-module framework [3] [5]. CI/CD workflows assume GitHub Actions.

Root Configuration Files

- `.editorconfig` (at `/`) standardizes formatting.

```
[*]
charset = utf-8
indent_style = space
indent_size = 4
end_of_line = crlf
insert_final_newline = true
trim_trailing_whitespace = true
```

- `README.md` (at `/README.md`) describes the project. A sample excerpt:

```
# AIUOKEEP (AI Upkeep Suite v2)
This repository implements **AI-assisted code quality enforcement** via a
Model Context Protocol (MCP) framework [6].  It coordinates PowerShell,
Python, and TypeScript quality tools behind guardrails, schema validation,
and OPA policies.  Key components include:
- **SafePatch pipeline:** multi-stage validation (format → lint → test →
SAST → policy) [7].
- **MCP servers:** wrappers around linters and scanners (PSScriptAnalyzer,
Ruff, Semgrep, etc.) [8] [9].
- **Guardrails:** pre-prompts, JSON schemas, Semgrep rules, and OPA
policies to enforce safe outputs [10] [11].
- **Audit & CI/CD:** GitHub Actions workflows run local `tools/Verify.ps1`,
with ledger logs and drift checks [12] [13].
For full architecture and setup, see [docs/ARCHITECTURE.md](/docs/
ARCHITECTURE.md).
```

This follows the structure and purpose outlined in the manifest [14].

- `CONTRIBUTING.md` (at `/CONTRIBUTING.md`): Development guidelines. Example content:

```
# Contributing
- Follow code conventions (`docs/conventions.md`) and naming rules [15] [11].
- Run `tools/Verify.ps1` locally to ensure all SafePatch stages pass [16].
- Include tests for any new functions (Pester for PowerShell, pytest for Python).
- Use pull requests: reference this guide and AIUOKEEP documentation.
```

- `.gitignore` (at `/.gitignore`): Ignore common build artifacts and sensitive files. Example entries:

```
# Generated files
*.pyc
*.log
/bin/
/obj/
/.runs/
/.mcp/temp/
# Secrets and config
*.keys
*.pem
```

- `LICENSE` (at `/LICENSE`): Project license, e.g. MIT or your chosen license. (Standard text, not shown here.)

MCP Configuration (`/.mcp`)

These scripts manage the MCP (Model Context Protocol) tool plane configuration [17] [18]. Each is a PowerShell function with typed parameters and extensive comments following [15] standards [3] [19]:

- `mcp_servers.json`: Authoritative list of MCP servers. Example structure:

```
{
  "servers": [
    {
      "name": "powershell",
      "type": "powershell",
      "address": "http://localhost:5001",
      "tools": ["PSScriptAnalyzer", "Pester"]
    },
    {
      "name": "python",
      "type": "python",
      "address": "http://localhost:5002",
      "tools": ["ruff", "black", "mypy", "pytest"]
    },
```

```
    {
      "name": "semgrep",
      "type": "sast",
      "address": "http://localhost:5003",
      "tools": ["semgrep"]
    },
    {
      "name": "secrets",
      "type": "secrets",
      "address": "http://localhost:5004",
      "tools": ["gitleaks"]
    },
    {
      "name": "policy",
      "type": "policy",
      "address": "http://localhost:5005",
      "tools": ["opa", "conftest"]
    }
  ]
}
```

- `access_groups.json`: Defines role-based access to tools. Example:

```
{
  "groups": {
    "reader": {
      "tools": ["ruff", "black", "PSScriptAnalyzer", "semgrep"]
    },
    "contributor": {
      "tools": ["ruff", "black", "PSScriptAnalyzer", "semgrep", "pytest",
"Pester"]
    },
    "maintainer": {
      "tools": ["ruff", "black", "PSScriptAnalyzer", "semgrep", "pytest",
"Pester", "gitleaks", "opa"]
    }
  }
}
```

- `Initialize-McpEnvironment.ps1`: Orchestrates environment setup (reads desired state and applies config) [20] [18]. Sample content:

```
<#
.SYNOPSIS
  Initialize or update the MCP server configuration.
#>
```

```powershell
[CmdletBinding(SupportsShouldProcess=$true)]
param(
  [Parameter(Mandatory)][string]$ConfigPath = ".mcp/mcp_servers.json",
  [Parameter(Mandatory)][string]$AccessPath = ".mcp/access_groups.json"
)

# Load desired server definitions
$desired = Get-Content -Raw -Path $ConfigPath | ConvertFrom-Json
# Load current state (could call Get-McpConfiguration)
$current = Get-McpConfiguration

# Merge configurations
$merged = New-McpConfigurationObject -Current $current -Desired $desired

if ($PSCmdlet.ShouldProcess("MCP Configuration", "Apply merged
configuration")) {
    Set-McpConfiguration -Configuration $merged
    Write-Verbose "MCP configuration updated."
}
```

- `Get-DesiredStateConfiguration.ps1`: Reads and validates desired state JSON. Stub:

```powershell
<#
.SYNOPSIS
  Reads MCP desired configuration from JSON and validates format.
#>
function Get-DesiredStateConfiguration {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory)][string]$Path
    )
    if (-Not (Test-Path $Path)) {
        throw "Configuration file not found: $Path"
    }
    $json = Get-Content -Raw -Path $Path
    try {
        $data = $json | ConvertFrom-Json
    } catch {
        throw "Invalid JSON in $Path: $_"
    }
    return $data
}
```

- `Get-McpConfiguration.ps1`: Retrieves current MCP state (e.g. from `.mcp/mcp.json` or live API). Example stub:

```
function Get-McpConfiguration {
    [CmdletBinding()]
    param(
        [string]$McpFile = ".mcp/mcp.json"
    )
    if (Test-Path $McpFile) {
        return Get-Content -Raw -Path $McpFile | ConvertFrom-Json
    } else {
        return @{ servers = @() }
    }
}
```

- `New-McpConfigurationObject.ps1`: Merges desired and current configs. Example:

```
function New-McpConfigurationObject {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory)][object]$Current,
        [Parameter(Mandatory)][object]$Desired
    )
    # Simple merge strategy: replace or add servers by name
    $merged = $Current
    foreach ($srv in $Desired.servers) {
        $existing = $merged.servers | Where-Object { $_.name -eq
$srv.name }
        if ($existing) {
            $existing | Remove-Variable
        }
        $merged.servers += $srv
    }
    return $merged
}
```

- `Set-McpConfiguration.ps1`: Writes the merged configuration (e.g. to `.mcp/mcp.json`). Example:

```
function Set-McpConfiguration {
    [CmdletBinding(SupportsShouldProcess=$true)]
    param(
        [Parameter(Mandatory)][object]$Configuration,
        [string]$McpFile = ".mcp/mcp.json"
    )
    if ($PSCmdlet.ShouldProcess("Set MCP config", "Write to $McpFile")) {
        $Configuration | ConvertTo-Json -Depth 5 | Out-File -FilePath
$McpFile -Encoding utf8
```

```
        Write-Verbose "MCP configuration saved to $McpFile"
    }
}
```

- `Test-McpEnvironment.ps1`: Validates server connectivity. Example:

```
function Test-McpEnvironment {
    [CmdletBinding()]
    param(
        [object]$Configuration
    )
    foreach ($srv in $Configuration.servers) {
        Write-Verbose "Testing MCP server '$($srv.name)' at $
($srv.address)..."
        try {
            $req = Invoke-WebRequest -Uri $srv.address -Method Head -
TimeoutSec 5
            Write-Host "Server $($srv.name) reachable." -ForegroundColor
Green
        } catch {
            Write-Warning "Cannot reach $($srv.name) at $($srv.address)."
        }
    }
}
```

MCP Server Implementations (`/mcp-servers`)

Wrappers exposing tools via MCP. We provide stubs illustrating entry points; actual implementations should call underlying analyzers. Example patterns:

- `ps_quality_mcp.ps1` (in `mcp-servers/powershell/`): Exposes PSScriptAnalyzer and Pester. A typical pattern is to define endpoints (e.g. via REST or named pipes). Below is a simplified listener using a hypothetical MCP PowerShell SDK:

```
<#
.SYNOPSIS
  MCP server for PowerShell quality (PSScriptAnalyzer, Pester).
#>
Import-Module PSScriptAnalyzer -ErrorAction Stop
Import-Module Pester -ErrorAction Stop

# Hypothetical: Register commands
Register-McpTool -Name "PSScriptAnalyzer" -ScriptBlock {
    param($filePath)
    Invoke-Formatter -Path $filePath -Recurse:$false -ErrorAction Stop |
```

```
        ConvertTo-Json
}
Register-McpTool -Name "Pester" -ScriptBlock {
    param($testFiles)
    $result = Invoke-Pester -Script $testFiles -PassThru
    $result | Select-Object Passed, Failed | ConvertTo-Json
}

# Start listener (example; actual method depends on MCP framework)
Start-McpServer -Port 5501
```

This follows the MCP server template in AIUOKEEP (wrapper scripts). Similar use of tool modules appears in the manifest [9] .

- `quality_mcp.py` (in `mcp-servers/python/`): Exposes Python tools (ruff, black, mypy, pytest). Example stub:

```
#!/usr/bin/env python3
import subprocess
import sys
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/run', methods=['POST'])
def run_tool():
    data = request.json
    tool = data.get('tool')
    files = data.get('files', [])
    if tool == "ruff":
        cmd = ["ruff", *files]
    elif tool == "black":
        cmd = ["black", "--check", *files]
    elif tool == "mypy":
        cmd = ["mypy", *files]
    elif tool == "pytest":
        cmd = ["pytest", "--json-report", "--json-report-file=report.json"]
    else:
        return jsonify(error="Unknown tool"), 400
    try:
        result = subprocess.run(cmd, capture_output=True, text=True,
check=True)
        return jsonify(output=result.stdout, error=result.stderr)
    except subprocess.CalledProcessError as e:
        return jsonify(output=e.output, error=e.stderr), 400
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5502)
```

This basic Flask server listens for JSON requests specifying `tool` and runs the command, returning JSON. In production, add authentication and error handling.

- `semgrep_mcp.py` (in `mcp-servers/sast/`): Wraps Semgrep SAST scanning:

```
#!/usr/bin/env python3
import subprocess, argparse, json

parser = argparse.ArgumentParser()
parser.add_argument("path", help="Directory or file to scan")
args = parser.parse_args()

# Run semgrep with the repository semgrep config
result = subprocess.run(
    ["semgrep", "--config", ".semgrep/semgrep.yml", "--config", ".semgrep/
semgrep-python.yml",
     "--config", ".semgrep/semgrep-powershell.yml", args.path],
    capture_output=True, text=True)
findings = result.stdout
print(json.dumps({"findings": findings}))
```

- `secrets_mcp.py` (in `mcp-servers/secrets/`): Uses gitleaks for secret scanning:

```
#!/usr/bin/env python3
import subprocess, sys, json

target = sys.argv[1] if len(sys.argv) > 1 else "."
result = subprocess.run(["gitleaks", "detect", "--source", target, "--
report-format", "json"],
                        capture_output=True, text=True)
try:
    leaks = json.loads(result.stdout)
    print(json.dumps(leaks))
except json.JSONDecodeError:
    print(json.dumps({"error": "Failed to parse gitleaks output"}))
```

- `policy_mcp.py` (in `mcp-servers/policy/`): Runs OPA/Conftest policies:

```
#!/usr/bin/env python3
import subprocess, sys, json
```

```
target = sys.argv[1] if len(sys.argv) > 1 else "."
result = subprocess.run(["conftest", "test", target], capture_output=True,
text=True)
print(json.dumps({"policy_results": result.stdout}))
```

Guardrail Schemas & Policies (`/policy` and `/.semgrep`)

The manifest lists several JSON schemas and OPA policies, plus Semgrep rules [21] [22]:

- `/policy/schemas/changeplan.schema.json`: Schema for AI ChangePlan outputs. Example:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "ChangePlan",
  "type": "object",
  "properties": {
    "changes": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "file": { "type": "string" },
          "diff": { "type": "string" },
          "type": { "type": "string", "enum": ["modify", "create",
"delete"] }
        },
        "required": ["file", "diff", "type"]
      }
    }
  },
  "required": ["changes"]
}
```

- `/policy/schemas/unifieddiff.schema.json`: Schema for diff format:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "UnifiedDiff",
  "type": "object",
  "properties": {
    "diff": { "type": "string" },
    "file": { "type": "string" }
  },
```

```
  "required": ["diff"]
}
```

- `/policy/opa/changeplan.rego`: OPA rule ensuring only allowed operations. Example rule:

```
package ai.changeplan

deny[reason] {
  input.change.file == ""
  reason := "File path is empty"
}
deny[reason] {
  input.change.type == "delete"
  reason := "Deletion not allowed by policy"
}
```

- `/policy/opa/forbidden_apis.rego`: Blocks dangerous calls (e.g. Invoke-Expression, eval). Example:

```
package ai.forbidden

forbidden[obj] {
  some i
  obj := input.diff[i]
  contains(obj, "Invoke-Expression")
}
```

- `/policy/opa/delivery_bundle.rego`: Ensures required files (tests, docs) exist:

```
package ai.delivery

violation[msg] {
  not input.files["README.md"]
  msg := "README.md is missing"
}
violation[msg] {
  not input.files["tests/"]
  msg := "Test suite is missing"
}
```

- `/.semgrep/semgrep.yml` (base rules): As provided [23] [24], e.g.:

```
rules:
  - id: audit-todo-comments
```

```
    message: "Resolve TODO/FIXME comments before delivering production
changes."
    languages: [python, typescript, javascript, powershell]
    severity: INFO
    pattern-regex: "(?i)\\b(TODO|FIXME)\\b"

  - id: disallow-http-urls
    message: "Use HTTPS endpoints. Plain HTTP is not permitted in
production."
    languages: [python, typescript, javascript]
    severity: ERROR
    pattern-regex: "http://"
  # ... (full content from semgrep.yml [23])
```

- `/.semgrep/semgrep-powershell.yml` (PowerShell rules): Include the user-provided rules [25] [26]:

```
rules:
  - id: powershell-no-write-host
    message: "Use Write-Output or structured logging instead of Write-
Host."
    severity: WARNING
    languages: [powershell]
    pattern: Write-Host ...
  - id: powershell-no-invoke-expression
    message: "Invoke-Expression bypasses guardrails and is not allowed."
    severity: ERROR
    languages: [powershell]
    pattern: Invoke-Expression ...
  - id: powershell-require-strictmode
    message: "Scripts must enable Set-StrictMode -Version Latest."
    severity: ERROR
    languages: [powershell]
    patterns:
      - pattern-not: |
          Set-StrictMode -Version Latest
      - pattern-either:
          - pattern: |
              function $FUNC(...) {
                  ...
              }
          - pattern: |
              [CmdletBinding()]
              param(...)
              ...
```

- `/.semgrep/semgrep-python.yml`: As provided [27] [28]:

```
rules:
  - id: python-no-eval
    message: "Avoid eval(); it allows arbitrary code execution."
    severity: ERROR
    languages: [python]
    pattern: eval(...)
  - id: python-no-exec
    message: "Avoid exec(); it allows arbitrary code execution."
    severity: ERROR
    languages: [python]
    pattern: exec(...)
  - id: python-shell-true
    message: "Do not run subprocesses with shell=True; use explicit
argument lists."
    severity: ERROR
    languages: [python]
    patterns:
      - pattern: subprocess.$FUNC(...)
      - metavariable-regex:
          metavariable: "$FUNC"
          regex: "(run|call|check_call|check_output|Popen)"
      - pattern-inside: |
          subprocess.$FUNC(..., shell=True, ...)
  - id: python-open-no-encoding
    message: "Specify encoding when opening text files for deterministic
results."
    severity: WARNING
    languages: [python]
    patterns:
      - pattern: open($FILENAME)
      - pattern-not: open($FILENAME, encoding=...)
      - pattern-not: open($FILENAME, $MODE, encoding=...)
```

- `/.semgrep/semgrep-secrets.yml`: As provided [29] [30]:

```
rules:
  - id: secret-aws-access-key
    message: "Potential AWS access key detected. Use secure secrets
storage."
    severity: ERROR
    languages: [python, typescript, javascript, generic]
    pattern-regex: "AKIA[0-9A-Z]{16}"
  - id: secret-aws-secret-key
    message: "Potential AWS secret key detected."
```

```
    severity: ERROR
    languages: [python, typescript, javascript, generic]
    pattern-regex: "(?i)aws(.{0,20})?(secret|access)_key(.{0,20})?=\\s*['\\
\"]?[A-Za-z0-9/+=]{40}"
  - id: secret-slack-token
    message: "Slack tokens must not be committed. Rotate the credential
immediately."
    severity: ERROR
    languages: [python, typescript, javascript, generic]
    pattern-regex: "xox[aboprs]-[0-9a-zA-Z-]{10,48}"
  - id: secret-private-key
    message: "Private key material detected."
    severity: ERROR
    languages: [generic]
    patterns:
      - pattern: "-----BEGIN PRIVATE KEY-----"
```

Code Quality Tools (`/tools`)

Configuration for linters and verification scripts [31] :

- `tools/PSScriptAnalyzerSettings.psd1` : PSScriptAnalyzer rules. Example:

```
@{
  IncludeRules = @('PSUseApprovedVerbs', 'PSUseSingularNouns')
  ExcludeRules = @('PSAvoidUsingWriteHost')
}
```

- `tools/Verify.ps1` : Local verification script (runs all checks). Example:

```
# Requires -MinimumVersion 7.0
[CmdletBinding()]
param()
Write-Host "Starting local verification (SafePatch) checks..."
Import-Module -Name Invoke-Build
Invoke-Build -Configuration @(
  'Invoke-FormatCheck',
  'Invoke-LintCheck',
  'Invoke-TypeCheck',
  'Invoke-UnitTests',
  'Invoke-SastScan',
  'Invoke-SecretScan',
  'Invoke-PolicyCheck'
)
```

- `tools/ruff.toml`: Ruff config for Python:

```
[tool.ruff]
line-length = 88
select = ["E", "W", "F"]
```

- `tools/mypy.ini`:

```
[mypy]
python_version = 3.12
strict = True
```

- `tools/pytest.ini`:

```
[pytest]
minversion = 6.0
addopts = --maxfail=1 --disable-warnings -q --cov=.
testpaths = tests
```

- `tools/.eslintrc.json`:

```
{
  "env": { "browser": true, "es2021": true },
  "extends": ["eslint:recommended", "plugin:@typescript-eslint/
recommended"],
  "parser": "@typescript-eslint/parser",
  "parserOptions": { "ecmaVersion": 12, "sourceType": "module" },
  "rules": { "indent": ["error", 2], "semi": ["error", "always"] }
}
```

- `tools/tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "outDir": "./dist"
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Validation Scripts (`/scripts/validation`)

Scripts orchestrating SafePatch stages. Each is a PowerShell function with proper `[CmdletBinding()]`, `-Verbose` logging, and uses guardrails (schema) where applicable [32] [33] :

- `Invoke-SafePatchValidation.ps1` : Runs full pipeline. Example:

```
<#
.SYNOPSIS
  Orchestrate full SafePatch validation pipeline.
#>
[CmdletBinding()]
param(
  [Parameter(Mandatory)][string]$WorkspacePath
)
function Invoke-SafePatch {
    Invoke-FormatCheck -Path $WorkspacePath
    Invoke-LintCheck -Path $WorkspacePath
    Invoke-TypeCheck -Path $WorkspacePath
    Invoke-UnitTests -Path $WorkspacePath
    Invoke-SastScan -Path $WorkspacePath
    Invoke-SecretScan -Path $WorkspacePath
    Invoke-PolicyCheck -Path $WorkspacePath
}
Invoke-SafePatch
```

- `Test-UnifiedDiff.ps1` : Validates diff format (using schema). Example stub:

```
function Test-UnifiedDiff {
    [CmdletBinding()]
    param([string]$DiffText)
    # Load JSON schema
    $schema = Get-Content -Raw 'policy/schemas/unifieddiff.schema.json' |
ConvertFrom-Json
    $diffObj = @{ diff = $DiffText }
    $validator = New-Object System.Text.Json.JsonSerializerOptions
    try {
        $json = $diffObj | ConvertTo-Json
        # Placeholder: actual JSON schema validation needed
        $true
    } catch {
        throw "Unified diff validation failed: $_"
    }
}
```

- `Test-ChangePlan.ps1`: Validates ChangePlan JSON and OPA rules. Example:

```
function Test-ChangePlan {
    [CmdletBinding()]
    param([string]$PlanPath)
    $plan = Get-Content -Raw $PlanPath
    $json = $plan | ConvertFrom-Json
    # Validate against JSON schema (similar to above)
    # Run OPA policy test (requires conftest)
    $opaResult = & conftest test --policy policy/opa $PlanPath
    if ($LASTEXITCODE -ne 0) {
        throw "ChangePlan policy check failed: $opaResult"
    }
}
```

- `Invoke-FormatCheck.ps1`: Runs formatters in check mode. Example:

```
function Invoke-FormatCheck {
    [CmdletBinding()]
    param([string]$Path)
    Write-Host "Running format checks..."
    # Python
    & black --check $Path
    & ruff --exit-zero --fix $Path
    # PowerShell
    Invoke-Formatter -Path $Path -Recurse -ErrorAction SilentlyContinue
}
```

- `Invoke-LintCheck.ps1`: Runs linters. Example:

```
function Invoke-LintCheck {
    [CmdletBinding()]
    param([string]$Path)
    Write-Host "Running lint checks..."
    # Python: ruff (with exit code)
    & ruff $Path
    # PowerShell: PSScriptAnalyzer
    Invoke-ScriptAnalyzer -Path $Path -Settings tools/
PSScriptAnalyzerSettings.psd1 |
        Out-String | Write-Host
}
```

- `Invoke-TypeCheck.ps1`: Runs type checkers. Example:

```
function Invoke-TypeCheck {
    [CmdletBinding()]
    param([string]$Path)
    Write-Host "Running type checks..."
    & mypy --config-file tools/mypy.ini $Path
    & tsc --project tools/tsconfig.json
}
```

- `Invoke-UnitTests.ps1`: Runs tests in sandbox. Example:

```
function Invoke-UnitTests {
    [CmdletBinding()]
    param([string]$Path)
    Write-Host "Running unit tests..."
    & pytest --config tools/pytest.ini --rootdir $Path
    Invoke-Pester -Script $Path\tests -OutputFormat NUnitXml
}
```

- `Invoke-SastScan.ps1`: Runs Semgrep. Example:

```
function Invoke-SastScan {
    [CmdletBinding()]
    param([string]$Path)
    Write-Host "Running SAST (Semgrep) scan..."
    & semgrep --config .semgrep/semgrep.yml --config .semgrep/semgrep-
python.yml --config .semgrep/semgrep-powershell.yml --timeout 60 $Path
}
```

- `Invoke-SecretScan.ps1`: Checks for secrets. Example:

```
function Invoke-SecretScan {
    [CmdletBinding()]
    param([string]$Path)
    Write-Host "Scanning for secrets..."
    & gitleaks detect --source $Path --exit-code 1
}
```

- `Invoke-PolicyCheck.ps1`: Applies OPA/Conftest policies. Example:

```
function Invoke-PolicyCheck {
    [CmdletBinding()]
    param([string]$Path)
    Write-Host "Checking policies (OPA/Conftest)..."
```

```
    & conftest test --policy policy/opa --no-fail $Path
}
```

Each script above is structured as a PowerShell function with typed parameters, help comments, and `Write-Host/Write-Verbose` logging, following [15] guidelines (e.g. using `-WhatIf` where applicable in state-change functions) [33] . Tests for these scripts would use Pester or pytest accordingly.

Sandbox Scripts (`/scripts/sandbox`)

Scripts to create isolated workspaces [34] :

- `scripts/sandbox/sandbox_linux.sh`: Sets up a Linux network namespace. Example stub:

```
#!/bin/bash
set -euo pipefail
echo "Creating isolated Linux sandbox..."
sudo ip netns add ai_sandbox
sudo ip link set lo netns ai_sandbox
sudo ip netns exec ai_sandbox bash -c "echo 'Sandbox ready'"
```

- `scripts/sandbox/sandbox_windows.ps1`: Uses Windows Firewall to restrict network. Example:

```
function New-NetworkRestrictedFirewall {
    New-NetFirewallRule -DisplayName "BlockAllOutbound_AI" -Direction
Outbound -Action Block -RemoteAddress Any -Protocol Any -Enabled True
    Write-Host "Windows sandbox firewall rules applied."
}
```

- `scripts/sandbox/New-EphemeralWorkspace.ps1`: Creates a temp Git worktree for validation. Example:

```
function New-EphemeralWorkspace {
    [CmdletBinding()]
    param([string]$BranchName = "temp-validation")
    git fetch origin
    git worktree add -d "temp_worktree" origin/$BranchName
    Write-Host "Created ephemeral workspace on branch $BranchName."
}
```

- `scripts/sandbox/Remove-EphemeralWorkspace.ps1`: Cleans up:

```
function Remove-EphemeralWorkspace {
    [CmdletBinding()]
    param([string]$WorkspacePath = "temp_worktree")
```

```
    git worktree remove $WorkspacePath
    git worktree prune
    Write-Host "Removed ephemeral workspace at $WorkspacePath."
}
```

Code Skeletons & Templates (`/templates`)

Standard templates for new modules and tests [35]:

- `templates/powershell/AdvancedFunction.ps1`: Defensive function template with StrictMode, ShouldProcess, typed output:

```
function Invoke-MyAdvancedOperation {
    [CmdletBinding(SupportsShouldProcess=$true, ConfirmImpact='Medium')]
    param(
        [Parameter(Mandatory)][string]$Target,
        [Parameter()][switch]$WhatIf
    )
    <#
    .SYNOPSIS
        Describe function purpose.
    .DESCRIPTION
        Longer description.
    .PARAMETER Target
        The target resource.
    .EXAMPLE
        Invoke-MyAdvancedOperation -Target "Name" -WhatIf
    #>
    Set-StrictMode -Version Latest
    if ($PSCmdlet.ShouldProcess($Target, 'MyAdvancedOperation')) {
        # Perform work here
        Write-Verbose "Processing target $Target..."
        return [PSCustomObject]@{ Target = $Target; Changed = $false }
    }
}
```

- `templates/powershell/Module.psm1`: PowerShell module stub:

```
# PowerShell module (.psm1)
$PSScriptRoot = Split-Path -Parent $MyInvocation.MyCommand.Definition
# Import public functions
. "$PSScriptRoot\Public\Invoke-MyAdvancedOperation.ps1"
# Export functions
Export-ModuleMember -Function Invoke-MyAdvancedOperation
```

- `templates/powershell/Module.psd1`: Module manifest stub:

```
@{
    RootModule = 'MyModule.psm1'
    ModuleVersion = '0.1'
    GUID = '00000000-0000-0000-0000-000000000000'
    Author = 'Your Name'
    Description = 'Module description'
    FunctionsToExport = @('Invoke-MyAdvancedOperation')
}
```

- `templates/powershell/Pester.Tests.ps1`: Example Pester test template:

```
Describe 'Invoke-MyAdvancedOperation' {
    It 'should not change state with -WhatIf' {
        { Invoke-MyAdvancedOperation -Target 'X' -WhatIf } | Should -Not -
Throw
    }
    It 'returns PSCustomObject with Target and Changed' {
        $res = Invoke-MyAdvancedOperation -Target 'X'
        $res | Should -BeOfType PSCustomObject
        $res.Target | Should -Be 'X'
        $res.Changed | Should -BeFalse
    }
}
```

- `templates/python/python_cli.py`: Python CLI template:

```
#!/usr/bin/env python3
import argparse
import logging

class Result:
    def __init__(self, success: bool, message: str):
        self.success = success
        self.message = message

def main():
    parser = argparse.ArgumentParser(description="My Python CLI tool.")
    parser.add_argument("--input", required=True, help="Input data")
    args = parser.parse_args()
    logging.basicConfig(level=logging.INFO)
    logging.info("Starting process")
    # TODO: implement logic
    res = Result(success=True, message="Completed successfully")
```

```
    print(res.message)

if __name__ == "__main__":
    main()
```

- `templates/python/test_template.py`: Pytest template:

```
import pytest
from my_module import MyFunction

def test_my_function_valid():
    input_data = "hello"
    output = MyFunction(input_data)
    assert output == "HELLO"

def test_my_function_invalid():
    with pytest.raises(ValueError):
        MyFunction(None)
```

- `templates/python/pyproject.toml`: Python project template:

```
[project]
name = "my_project"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
dependencies = []
[tool.pytest.ini_options]
minversion = "6.0"
addopts = "--verbose"
testpaths = ["tests"]
```

- `templates/typescript/typescript_module.ts`: TypeScript stub:

```
export function myFunction(input: string): string {
  return input.toUpperCase();
}
// Example usage:
// console.log(myFunction("test"));
```

These templates follow the **5-module strategy** (data acquisition, transformation, state change, validation, orchestration) with clear naming and contracts [3] [36]. For instance, functions use PascalCase and Verb-Noun (per [15†L39-L47]), include comment-based help, support `-WhatIf` for mutations, and return typed PSCustomObject results [33].

Pre-commit Configuration

- `.pre-commit-config.yaml` (at `/`): Runs local validators on commit. Example:

```
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.3.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
  - repo: https://github.com/deten/improved-pre-commit-hooks
    rev: v0.0.3
    hooks:
      - id: python-check-utf8-encoding
```

- `scripts/hooks/install-hooks.ps1`: Installs git hooks (calls `pre-commit` or copies scripts):

```
# Requires pre-commit installed
Write-Host "Installing pre-commit hooks..."
& pre-commit install
Write-Host "Hooks installed."
```

- `scripts/hooks/pre-commit.ps1`: Additional logic, if any. Stub:

```
#!/usr/bin/env pwsh
# Custom pre-commit logic (invoked by pre-commit, for example)
Write-Host "Running custom pre-commit checks..."
.\tools\Verify.ps1
```

CI/CD Workflows (`/.github/workflows`)

Example GitHub Actions workflows (using composite and reuse patterns [37]):

- `/.github/workflows/quality.yml`: Main quality gate (invokes local `Verify.ps1`). Example:

```
name: Quality Check
on: [push, pull_request]
jobs:
  verify:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with: {python-version: '3.12'}
```

```
      - uses: actions/setup-dotnet@v3
        with: {dotnet-version: '6.0.x'}
      - name: Install Pre-commit
        run: pip install pre-commit
      - name: Run pre-commit
        run: pre-commit run --all-files
      - name: Run Verify.ps1
        uses: git-powershell@v1
        with:
          script: |
            pwsh -File tools/Verify.ps1
```

- `powershell-verify.yml`: Windows PowerShell job:

```
name: PowerShell Verify
on: [pull_request]
jobs:
  ps-lint:
    runs-on: windows-latest
    steps:
      - uses: actions/checkout@v3
      - name: Install modules
        run: Install-Module PSScriptAnalyzer -Force
      - name: PSScriptAnalyzer
        run: Invoke-ScriptAnalyzer -Path . -Recurse
      - name: Pester Tests
        run: Invoke-Pester -Script tests
```

- `python-verify.yml`:

```
name: Python Verify
on: [pull_request]
jobs:
  lint-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Python
        uses: actions/setup-python@v4
        with: {python-version: '3.12'}
      - run: pip install ruff black mypy pytest
      - run: black --check .
      - run: ruff .
      - run: mypy .
      - run: pytest --maxfail=1 --disable-warnings -q
```

- `typescript-verify.yml`:

```
name: TypeScript Verify
on: [pull_request]
jobs:
  ts-lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with: {node-version: '18.x'}
      - run: npm install
      - run: npx eslint .
      - run: npx tsc --noEmit
```

- `sast-secrets.yml`:

```
name: SAST and Secrets Scan
on: [push]
jobs:
  scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run Semgrep
        run: semgrep --config .semgrep/semgrep.yml .
      - name: Run Secrets Scan (gitleaks)
        run: gitleaks detect --source . --no-git
```

- `policy-check.yml`:

```
name: Policy Check
on: [push]
jobs:
  conftest:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: instrumenta/conftest-action@v1
        with:
          rule_path: policy/opa
```

- `drift-detection.yml` (nightly): Checks for config drift.

```yaml
name: Drift Detection
on:
  schedule:
    - cron: '0 0 * * 0'
jobs:
  drift:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Check GitHub branch protection
        run: ./.github/scripts/check-branch-protection.ps1
      - name: Check policy drift
        run: ./.github/scripts/check-policy-drift.ps1
```

- `renovate.json`: Bot config for dependency updates:

```
{
  "extends": ["config:base"],
  "schedule": ["before 5am on Monday"]
}
```

These workflows reuse known actions (e.g. checkout, setup, eslint) [37] to avoid reinventing processes.

Audit & Observability Scripts (`/scripts/audit`)

- `scripts/audit/New-RunLedgerEntry.ps1`: Creates signed JSONL ledger entry. Example:

```
function New-RunLedgerEntry {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory)][string]$User,
        [Parameter(Mandatory)][string]$Action,
        [Parameter(Mandatory)][string]$Details
    )
    $entry = @{
        Timestamp = (Get-Date).ToString("o")
        User = $User
        Action = $Action
        Details = $Details
    }
    $json = $entry | ConvertTo-Json
    $signature = (Get-FileHash -InputStream
([System.IO.MemoryStream]::new([System.Text.Encoding]::UTF8.GetBytes($json)))
-Algorithm SHA256).Hash
    $ledgerLine = "$json|$signature"
```

```
    Add-Content -Path ".runs/ledger.jsonl" -Value $ledgerLine
    Write-Host "Ledger entry created."
}
```

- `scripts/audit/Get-RunLedger.ps1`: Queries ledger entries (e.g. by user or date). Example stub:

```
function Get-RunLedger {
    [CmdletBinding()]
    param([string]$Filter)
    $lines = Get-Content -Path ".runs/ledger.jsonl"
    $entries = foreach ($line in $lines) {
        $parts = $line -split "\|"
        $data = $parts[0] | ConvertFrom-Json
        [PSCustomObject]@{
            Timestamp = $data.Timestamp
            User = $data.User
            Action = $data.Action
            Details = $data.Details
            Hash = $parts[1]
        }
    }
    if ($Filter) { $entries = $entries | Where-Object { $_.User -eq
$Filter } }
    return $entries
}
```

- `scripts/audit/Export-WeeklyReport.ps1`: Summarizes metrics. Example stub:

```
function Export-WeeklyReport {
    [CmdletBinding()]
    param([string]$Output = "reports/weekly_report.txt")
    $weekEntries = Get-RunLedger | Where-Object {
        [datetime]$_ .Timestamp -gt (Get-Date).AddDays(-7)
    }
    $summary = $weekEntries | Group-Object -Property Action | Select Name,
Count
    $report = "Weekly Report - $(Get-Date -Format yyyy-MM-dd)" + "`n"
    foreach ($item in $summary) {
        $report += "$($item.Name): $($item.Count) occurrences`n"
    }
    $report | Out-File -FilePath $Output
    Write-Host "Weekly report saved to $Output"
}
```

- `scripts/audit/Invoke-DriftDetection.ps1`: Checks for guardrail drift (e.g., missing policy rules). Example stub:

```
function Invoke-DriftDetection {
    [CmdletBinding()]
    param()
    Write-Host "Checking for policy drift..."
    # Example: compare local semgrep.yml vs. a golden version
    $current = Get-Content .semgrep/semgrep.yml
    $golden = Get-Content docs/seed/semgrep.yml
    if ($current -ne $golden) {
        Write-Warning "Semgrep configuration has drifted from golden copy."
    }
}
```

- `schemas/ledger.schema.json`: JSON schema for ledger entries:

```
{
  "title": "RunLedgerEntry",
  "type": "object",
  "properties": {
    "Timestamp": { "type": "string", "format": "date-time" },
    "User": { "type": "string" },
    "Action": { "type": "string" },
    "Details": { "type": "string" },
    "Hash": { "type": "string" }
  },
  "required": ["Timestamp", "User", "Action", "Hash"]
}
```

Database Schema (`/database`)

- `schema.sql`: Defines tables for access groups, tools, policies, ledger. Example:

```
CREATE TABLE AccessGroups (
  id SERIAL PRIMARY KEY,
  name TEXT UNIQUE NOT NULL
);
CREATE TABLE Tools (
  id SERIAL PRIMARY KEY,
  name TEXT UNIQUE NOT NULL,
  group_id INT REFERENCES AccessGroups(id)
);
CREATE TABLE Policies (
  id SERIAL PRIMARY KEY,
```

```
  name TEXT UNIQUE NOT NULL,
  definition JSONB NOT NULL
);
CREATE TABLE LedgerEntries (
  id SERIAL PRIMARY KEY,
  entry_time TIMESTAMP NOT NULL,
  user TEXT NOT NULL,
  action TEXT NOT NULL,
  details JSONB,
  signature TEXT NOT NULL
);
```

- `seed_data.sql`: Inserts default groups/policies:

```
INSERT INTO AccessGroups (name) VALUES ('reader'), ('contributor'),
('maintainer');
INSERT INTO Tools (name, group_id) VALUES
  ('ruff', 1), ('black', 1), ('PSScriptAnalyzer', 1),
  ('semgrep', 1), ('pytest', 2), ('Pester', 2),
  ('gitleaks', 3), ('opa', 3);
INSERT INTO Policies (name, definition) VALUES
  ('forbidden_apis', '{}'::jsonb), ('delivery_bundle', '{}'::jsonb);
```

- `migrations/`: Directory for future schema migrations (e.g. `V2__add_new_table.sql`).

File Routing System (`/file-routing`)

- `file-routing/file_router.config.json`: Maps project codes to dirs. Example:

```
{
  "PROJECT1": "modules/phase_1_intake_routing",
  "PROJECT2": "modules/phase_2_discovery_scoring"
}
```

- `file-routing/FileRouter_Watcher.ps1`: Watches Downloads folder and moves files by naming convention (pattern-driven). Example snippet:

```
$watcher = New-Object System.IO.FileSystemWatcher
$watcher.Path = "$env:USERPROFILE\Downloads"
$watcher.Filter = "*.*"
$watcher.EnableRaisingEvents = $true
$onChange = Register-ObjectEvent $watcher 'Created' -Action {
    param($sender, $event)
    Start-Sleep -Milliseconds 500
    $file = $event.FullPath
```

```
    # Determine target path based on naming (PROJECT-AREA-..._timestamp_)
    $name = Split-Path $file -Leaf
    if ($name -match '^(?<code>\w+)-(?<area>\w+)_') {
        $proj = $matches['code']
        $targetDir = (Get-Content file_router.config.json | ConvertFrom-
Json).$proj
        Move-Item $file -Destination $targetDir
    }
}
```

- `file-routing/Naming_Convention_Guide.md`: Documents naming. Example excerpt:

```
# File Naming Convention
Files must follow: `PROJECT-AREA-
<SUBAREA>_name_YYYYMMDDHHMMSS_v<version>_<ULID>_<SHA8>`.
- **PROJECT**: 4-letter project code.
- **AREA**: Functional area code (matching module registry).
- **SUBAREA**: Optional subfolder code.
- **name**: Descriptive name.
- **timestamp**: UTC time.
- **version**: Version number (e.g., v1).
- **ULID**: Global unique identifier.
- **SHA8**: First 8 chars of content SHA256.
```

This aligns with the deterministic naming discussed in the manifest and plans.

Edit Engine (`/tools/edit-engine`)

Utilities for applying patches/diffs:

- `tools/edit-engine/apply_patch.ps1`: Applies a unified diff. Example stub:

```
function Apply-UnifiedPatch {
    [CmdletBinding()]
    param([string]$DiffFile, [string]$TargetDir)
    git apply --directory=$TargetDir $DiffFile
    if ($LASTEXITCODE -ne 0) { throw "Patch failed to apply." }
    Write-Host "Patch applied successfully."
}
```

- `tools/edit-engine/apply_jsonpatch.ps1`: Applies JSON Patch (RFC 6902). Example stub:

```
function Apply-JsonPatch {
    [CmdletBinding()]
    param([string]$JsonFile, [string]$PatchFile)
```

```
    $json = Get-Content -Raw $JsonFile | ConvertFrom-Json
    $patch = Get-Content -Raw $PatchFile | ConvertFrom-Json
    # Use .NET JsonPatch if available; else stub:
    foreach ($op in $patch) {
        if ($op.op -eq "replace") {
            $path = $op.path.TrimStart('/')
            $json.$path = $op.value
        }
    }
    $json | ConvertTo-Json | Set-Content $JsonFile
    Write-Host "JSON patch applied."
}
```

- `tools/edit-engine/run_comby.ps1`: Applies structural refactors with Comby:

```
function Invoke-Comby {
    [CmdletBinding()]
    param([string]$Pattern, [string]$Replacement, [string]$Path)
    & comby $Pattern $Replacement $Path --matcher="generic"
    Write-Host "Comby transformation applied."
}
```

- `tools/edit-engine/run_ast_mod.ps1`: Applies PowerShell AST transformations:

```
function Invoke-AstTransform {
    [CmdletBinding()]
    param([string]$ScriptPath, [scriptblock]$Transform)
    $ast =
[System.Management.Automation.Language.Parser]::ParseFile($ScriptPath,
[ref]$null, [ref]$null)
    $modifiedAst = & $Transform $ast
    # Convert AST back to code (requires AST to script conversion)
}
```

- `tools/edit-engine/regenerate.ps1`: Regenerates files from templates:

```
function Invoke-Regenerate {
    [CmdletBinding()]
    param([string]$ModuleName, [hashtable]$Parameters)
    # Example: expand templates
    Get-ChildItem -Path templates -Recurse | ForEach-Object {
        $dest = $_.FullName -replace '\\templates\\', "\\modules\\
$ModuleName\\"
        Copy-Item $_.FullName $dest
        # Replace placeholders in $dest with $Parameters values
```

```
    }
    Write-Host "Files regenerated for module $ModuleName."
}
```

Documentation (`/docs`)

Key documentation files [38]:

- `docs/ARCHITECTURE.md`: Detailed architecture. Should describe SafePatch flow, MCP tool plane, guardrail layers. For example, copy relevant sections from **AIUOKEEP.md** [39] [13] and expand. E.g., explain that "Agents generate a ChangePlan, which is schema-validated [40] and then applied in an isolated sandbox through the SafePatch pipeline [41]."
- `docs/GUARDRAILS.md`: Policy rationale. Can incorporate content from the guardrail framework [42] [43].
- `docs/MCP_INTEGRATION.md`: Guide to adding or configuring MCP servers. Example outline:

```
# MCP Integration Guide
1. Define server in `/.mcp/mcp_servers.json` (see example).
2. Implement tool wrappers (scripts in `mcp-servers/`) exposing the
analysis tools via HTTP/CLI.
3. Test connectivity with `Get-McpConfiguration` and `Test-
McpEnvironment.ps1` [18].
4. Assign access in `access_groups.json` and reload with `Initialize-
McpEnvironment.ps1`.
```

- `docs/VALIDATION_PIPELINE.md`: Describes SafePatch stages. Example bullet steps: format, lint, etc. [7].
- `docs/AGENT_GUIDELINES.md`: Best practices for AI agents. Copy or refer to **AGENT_GUIDELINES.md** content [44] [45] (already written).
- `docs/TROUBLESHOOTING.md`: Common fixes. E.g.:

```
# Troubleshooting
- **PSScriptAnalyzer fails**: Run `Update-Module PSScriptAnalyzer`, ensure
the correct ruleset is loaded.
- **Tests failing unexpectedly**: Check sandbox isolation; tests may
require network or external data.
- **Semgrep issues**: Verify `.semgrep/*.yml` rules syntax. Use `semgrep --
validate`.
- **Policy denials**: Inspect Conftest output. If a guardrail is too
strict, update schemas or policies accordingly [11].
```

- `docs/conventions.md`: Code conventions. Summarize rules from the Module Template Pack [3]. Example:

```
# Code Conventions
- **Naming:** Use PascalCase Verb-Noun for functions (e.g. `Get-UserData`, `Convert-DataFormat`) [3].
- **Modules:** Export only necessary public functions; use comment-based help.
- **Error Handling:** Use `try/catch` and `throw` for failures; no unhandled script errors.
- **Idempotence:** State-change functions support `-WhatIf/-Confirm` [33]; acquisition/transformation functions must not alter state [33].
- **Logging:** Use `Write-Verbose`/`Write-Information` for diagnostics; no printing to console except final results.
- **Python:** Follow PEP 8; type hints for all functions; use `pytest` for tests.
- **TypeScript:** Enable `strict` mode in `tsconfig.json`; include JSDoc comments.
```

Testing Infrastructure (`/tests`)

Directories for test assets and suites [46]:

- `tests/fixtures/`: Include sample data and expected outputs for tests (JSON files, example code).
- `tests/integration/`: End-to-end tests (e.g. apply a diff, run full SafePatch, verify results).
- `tests/unit/`: Unit tests for individual functions. For example, tests for each module function created via the templates.
- `tests/Invoke-IntegrationTests.ps1`: Runs all integration tests. Example:

```
function Invoke-IntegrationTests {
    [CmdletBinding()]
    param()
    Write-Host "Running integration test suite..."
    # Possibly spin up sandbox, apply patches, validate outcomes.
    # Placeholder:
    Write-Host "Integration tests not yet implemented."
}
```

Summary

All files above are organized per the AIUOKEEP manifest [1] [47] and use battle-tested patterns [3] [48]. We've adhered to modular design and guardrail rules: functions have clear responsibilities and contracts [5] [36]. For example, state-change cmdlets use `[CmdletBinding(SupportsShouldProcess=$true)]` and return structured PSCustomObjects [33]. The structure supports the SafePatch pipeline and audit trail described in the architecture [13] [49].

Each script should be accompanied by Pester (PowerShell) or pytest (Python) tests as indicated. Following [15] and [13] ensures thorough testing: e.g., validation modules return `{Passed, Message}` objects, transformation functions are pure with deterministic outputs, etc. These patterns (e.g. debounced file watching, Invoke-Build orchestration) were identified as proven in existing tools [48] [3].

Overall, this code layout provides a ready-to-use pipeline: AI agents propose diffs, the **ChangePlan** is schema-checked and policy-checked, then the patches are applied in a sandbox and validated through format, lint, test, SAST, and policy gates [7] [41] before merging. The guardrails and documentation (AGENT_GUIDELINES, GUARDRAILS, etc.) ensure safe, deterministic operation [44] [11].

Sources: AIUOKEEP architecture and manifest [1] [50]; modular code patterns [3] [5]; guardrail guidelines [44] [11].

---

[1] [2] [6] [7] [8] [9] [10] [12] [13] [14] [17] [18] [20] [21] [22] [31] [34] [35] [38] [39] [40] [41] [46] [47] [50] AIUOKEEP.md
file://file-9DJ5a7g8dd2qXiW6N7VhBY

[3] [19] [32] [33] [36] Module Template Pack (PowerShell)_CODE_MODULAZTION.md
file://file-5GPJhNuf6azVUMP7vXVgri

[4] [5] The Engineering Framework for MODULAR CODE.docx
file://file-5QmATeqUonhRcni4Q9oLSE

[11] [42] [43] [49] GUARDRAILS.md
file://file-DCKok4D4hctd6pDCdNEdU4

[15] [16] [44] [45] AGENT_GUIDELINES.md
file://file-ToUgyKAoJHAJUV5x6Y1WiU

[23] [24] semgrep.yml
file://file-MyJBMLDv9mxZJPSmEvZ12N

[25] [26] semgrep-powershell.yml
file://file-3AR24roRJGxvDjeKbChyWv

[27] [28] semgrep-python.yml
file://file-TC67CDiE5YEpbijiLRcy5e

[29] [30] semgrep-secrets.yml
file://file-LWo5jHp9Qf278EgQxaXVXm

[37] [48] Proven DevOps Patterns for AI-Operated Pipeline.pdf
file://file-8oaw3zJeWqGHJqHM2xvCio