

Simulink® Check™

Support Package for CI/CD Automation for Simulink®
Check™ User's Guide



MATLAB® & SIMULINK®

R2024b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

CI/CD Automation for Simulink® Check™ User's Guide

© COPYRIGHT 2022-2024 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2024 Online only New for Version 24.2 (R2024b)

1	Run Tasks with Process Advisor	
	Automate and Run Tasks with Process Advisor	1-2
	View and Modify Default Process	1-2
	Run Tasks and Review Results	1-6
	Identify Impact of Changes	1-8
	Rerun Impacted Tasks with Incremental Build	1-10
	Export Build Report	1-11
	Explore Other Options	1-11
	Programmatically Run Tasks	1-13
	Run Pipeline of Tasks	1-13
	View Available Tasks Iterations	1-14
	Generate Build Report	1-14
	Specify Settings for Process Advisor and Build System	1-16
	Project Settings	1-16
	User Settings	1-18
	Locally Reproduce Issues Found in CI	1-20
	Get Latest Project Files	1-20
	Download and Copy CI Artifacts into Project	1-20
	Debug in Process Advisor	1-20
2	Customize Your Process Model	
	Modify Default Process Model to Fit Your Process	2-2
	Open Project	2-2
	Create Process Model	2-2
	Inspect Default Process Model	2-2
	Section A — Add or Remove Built-In Tasks	2-4
	Section B — Modify Behavior of Built-In Tasks	2-6
	Section C — Specify Dependencies Between Tasks	2-7
	Section D — Specify Preferred Task Execution Order	2-7
	Overview of Process Model	2-9
	Process Model	2-9
	Tasks	2-9
	Queries	2-11
	Use Your Process	2-12

Add Tasks to Process	2-13
Open Process Model	2-13
Add Tasks	2-13
Built-In Tasks	2-14
Custom Tasks	2-15
Reconfigure Task Behavior	2-17
Open Process Model	2-17
Task Inputs	2-17
Task Action	2-18
Task Iterations	2-19
Define Task Relationships	2-21
Open Process Model	2-21
Specify Relationships	2-21
Find Artifacts with Queries	2-23
Built-In Queries	2-23
Custom Queries	2-25
Dynamically Resolve Paths with Tokens	2-27
Create Custom Tasks	2-28
Custom Task that Runs Existing Script	2-28
Custom Task for Specialized Functionality	2-29
Example Custom Tasks	2-33
Create Custom Queries	2-39
Choose Superclass for Custom Query	2-39
Define and Use Custom Query in Process	2-39
Example Custom Queries	2-41
Group Tasks with Subprocesses	2-46
Open Process Model	2-46
Add Tasks to Specific Subprocess	2-46
Considerations for Subprocess Boundaries	2-47
Example Process Model with Subprocesses	2-48
Manage Multiple Build and Verification Workflows Using Processes ...	2-49
Open Process Model	2-49
Overview of Processes	2-49
Define New Processes	2-50
Use Specific Process	2-54
Best Practices for Process Model Authoring	2-56
Keep Process Model File in Project Root	2-56
Make Sure Only One Process Model File on Path	2-56
Review Untracked Dependencies	2-56
Share Queries Across Tasks	2-56
Exclude Files from Change Tracking in Process Advisor	2-59
Process Model	2-59
Task Inputs	2-60
Task Outputs	2-60
Handling Untracked Dependencies	2-62

Test Tasks and Queries	2-63
Open Project	2-63
Find Artifacts Using Query	2-63
Run Task for Specific Artifacts	2-64
Dry Run Tasks to Test Process Model	2-66
Dry Run Tasks	2-66
Dry Run Results	2-66
Specify Dry Run Functionality for Tasks	2-67
Troubleshoot Missing Tasks, Artifacts, and Dependencies	2-70
Artifact Issues	2-70
Project Analysis Issues	2-70
Limitations on Incremental Build	2-72
Other Limitations	2-73
Handling Invalid Dependencies	2-74
Analyze Project From Scratch	2-76

Integrate Process into CI

3

Approaches to Running Processes in CI	3-2
Before You Integrate	3-2
GitHub	3-2
GitLab	3-3
Jenkins	3-3
Other Platforms	3-4
Integrate Process into GitHub	3-5
Set Up GitHub Project and Runner	3-5
Connect MATLAB Project to GitHub	3-5
Generate Pipeline Configuration File	3-6
Use Pipeline Configuration File in GitHub Actions Workflow	3-6
Integrate Process into GitLab	3-8
Set Up GitLab Project and Runner	3-8
Connect MATLAB Project to GitLab	3-9
Configure Template to use GitLab Runner	3-9
Make Optional Customizations	3-10
Generate Pipeline in GitLab	3-11
Optional Customizations	3-10
Integrate Process into Jenkins	3-14
Set Up Jenkins	3-14
Connect Jenkins Project to Repository	3-15
Configure and Use Jenkinsfile Template	3-15
Make Optional Customizations	3-17
Generate Pipeline in Jenkins	3-18
Integrate Process into Other CI Platforms	3-19
Before You Integrate	3-19
Run MATLAB in Batch Mode	3-19

How Pipeline Generation Works	3-21
Summary of Support	3-21
Generated Pipelines	3-22
Optional Pipeline Customization	3-22
Parallel Pipeline Architectures	3-24
Tips for Setting Up CI Agents	3-28
Product Installation	3-28
Dry Run Your Process	3-28
Set Up Virtual Display Machines Without Displays	3-29
Create Docker Container for Support Package	3-30
Best Practices for Effective Builds	3-32
Use Incremental Builds for Regular Submissions	3-32
Run Full Builds for Qualifying Software	3-32
Cache Models and Other Artifacts Used During Build	3-32

Version History

4

September 2024	4-2
Documentation	4-2
Features	4-2
July 2024	4-5
Features	4-5
June 2024	4-7
Features	4-7
May 2024	4-10
Features	4-10
April 2024	4-13
March 2024	4-14
Features	4-14
February 2024	4-20
Features	4-20
December 2023	4-23
November 2023	4-25
October 2023	4-27
September 2023	4-29
August 2023	4-31

July 2023	4-32
June 2023	4-33
April 2023	4-36
March 2023	4-39
February 2023	4-40
December 2022	4-41
November 2022	4-42
October 2022	4-43
September 2022	4-44
August 2022	4-45

Run Tasks with Process Advisor

- “Automate and Run Tasks with Process Advisor” on page 1-2
- “Programmatically Run Tasks” on page 1-13
- “Specify Settings for Process Advisor and Build System” on page 1-16
- “Locally Reproduce Issues Found in CI” on page 1-20

Automate and Run Tasks with Process Advisor

You can automate common tasks in your model-based development and verification workflow by using the CI/CD Automation for Simulink Check support package. The support package contains a default process model with built-in tasks for performing common activities such as checking modeling standards with Model Advisor, running tests with Simulink Test™, and generating code with Embedded Coder®. You can view, edit, and run these tasks from the Process Advisor app or run tasks by using the Process Advisor function `runprocess`.

This example shows how to:

- View and modify the default process model to fit your development process.
- Run tasks and review results by using Process Advisor.
- Identify the impact of a change and incrementally rerun impacted tasks.
- Export a build report that summarizes the task results.

View and Modify Default Process

You can define development and verification processes for a project by using a script called a process model. Process Advisor automatically reads the process model and uses the file to determine which tasks to run, how the tasks perform their actions, and the order in which tasks run. If your project does not already have a process model, Process Advisor automatically opens a default process model file that uses built-in tasks to perform common model-based design activities. You can edit the process model to add, remove, or reconfigure the tasks in the process. For information on how Process Advisor generates pipelines of tasks based on the process model, tasks, and queries, see “Overview of Process Model” on page 2-9.

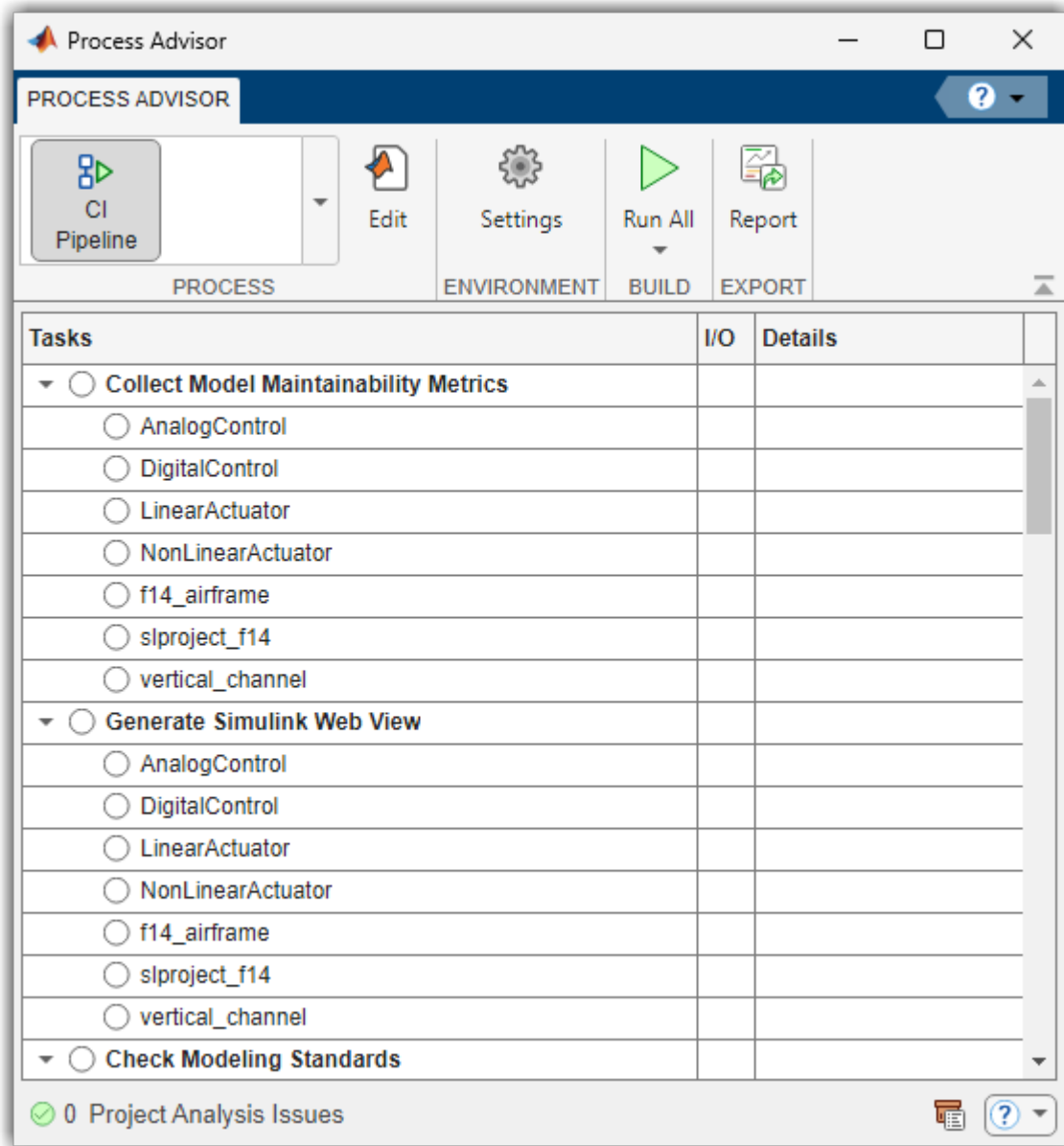
- 1 Open a project that contains your files. If you do not have a project, see “Create Projects” or open an example project by entering:


```
openExample('simulink/UsingAProjectExample')
```

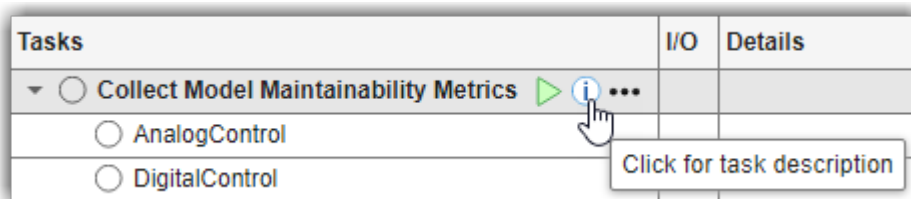
- 2 Open Process Advisor. On the **Project** tab, in the **Tools** section, click **Process Advisor** or enter `processAdvisorWindow` at the command line.


Process Advisor opens in a standalone window. If Process Advisor needs to perform an initial analysis on your project, the app shows the Enable Artifact Tracing dialog box. Click **Enable and Continue**.

The tasks defined in the process model appear in the **Tasks** column in Process Advisor. The default process model contains built-in tasks for several common tasks like checking modeling standards with Model Advisor, running tests with Simulink Test, and generating code with Embedded Coder.



- 3 To view information about a task, point to the task in the **Tasks** column and click the information icon .



- 4 You can add or remove tasks from the process by editing the process model. Inspect and edit the process model by clicking the **Edit** button  in the toolbar. Process Advisor opens the process model file, `processmodel.m`, in the MATLAB® Editor.

You can add or remove built-in tasks from the process by setting the variable associated with a built-in task to true or false. For example, to add the built-in task for design error detection to your process, you can update the code to set includeDesignErrorDetectionTask to true.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Include/Exclude Tasks in processmodel
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

includeModelMaintainabilityMetricTask = true;
includeModelTestingMetricTask = true;
includeModelStandardsTask = true;
includeDesignErrorDetectionTask = true;
includeFindClones = true;
includeModelComparisonTask = false;
includeSDDTask = true;
includeSimulinkWebViewTask = true;
includeTestsPerTestCaseTask = true;
includeMergeTestResultsTask = true;
includeGenerateCodeTask = true;
includeAnalyzeModelCode = true && exist('polyspaceroot','file');
includeProveCodeQuality = true && (~isempty(ver('pscodeprover')) ...
    || ~isempty(ver('pscodeproverserver')));
includeCodeInspection = false;
    
```

Each variable is associated with a task in the default process model.

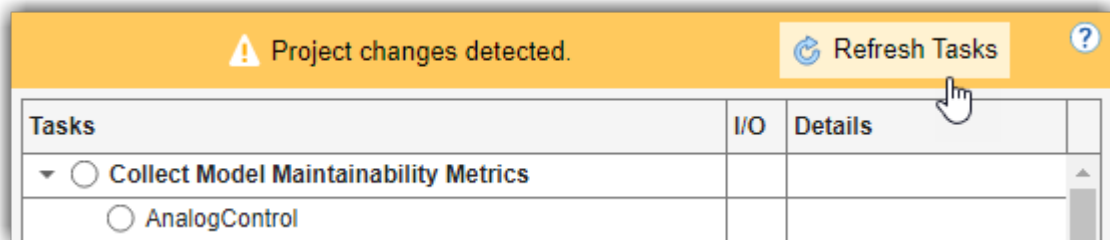
Variable in Default Process Model	Task Title	Built-In Task
includeModelMaintainabilityMetricTask	Collect Model Maintainability Metrics	padv.builtin.task.CollectMetrics
includeModelTestingMetricTask	Collect Model Testing Metrics	padv.builtin.task.CollectMetrics
includeModelStandardsTask	Check Modeling Standards	padv.builtin.task.RunModelStandards
includeDesignErrorDetectionTask	Detect Design Errors	padv.builtin.task.DetectDesignErrors
includeFindClones	Find Clones	padv.builtin.task.FindClones
includeModelComparisonTask	Generate Model Comparison	padv.builtin.task.GenerateModelComparison
includeSDDTask	Generate SDD Report	padv.builtin.task.GenerateSDDReport
includeSimulinkWebViewTask	Generate Simulink Web View	padv.builtin.task.GenerateSimulinkWebView
includeTestsPerTestCaseTask	Run Tests	padv.builtin.task.RunTestsPerTestCase
includeMergeTestResultsTask	Merge Test Results	padv.builtin.task.MergeTestResults
includeGenerateCodeTask	Generate Code	padv.builtin.task.GenerateCode

Variable in Default Process Model	Task Title	Built-In Task
includeAnalyzeModelCode	Check Coding Standards	padv.builtin.task.AnalyzeModelCode
includeProveCodeQuality	Prove Code Quality	padv.builtin.task.AnalyzeModelCode
includeCodeInspection	Inspect Code	padv.builtin.task.RunCodeInspection

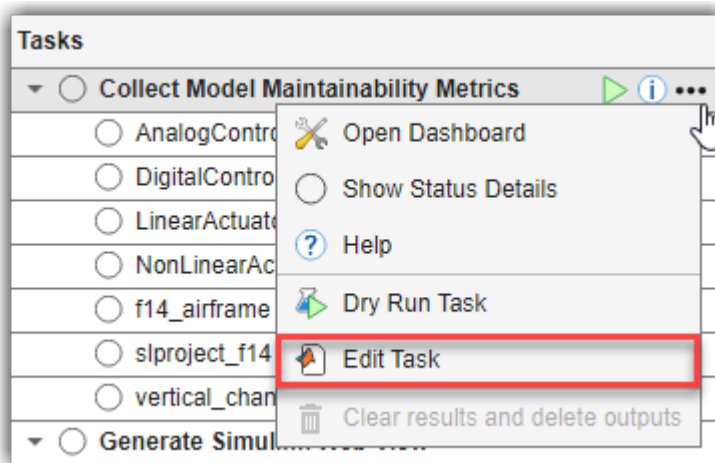
For more information on how to get started with the default process model, see “Modify Default Process Model to Fit Your Process” on page 2-2.

- After you finish making changes to the process model, you can view the updated list of tasks in Process Advisor by returning to the Process Advisor window and, in the warning banner, clicking **Refresh Tasks**.

Process Advisor refreshes to reflect the updated process model. The **Tasks** column now includes a task for **Detect Design Errors**.



- You can view the source code for a built-in task by pointing to the task and clicking ... > **Edit Task**.

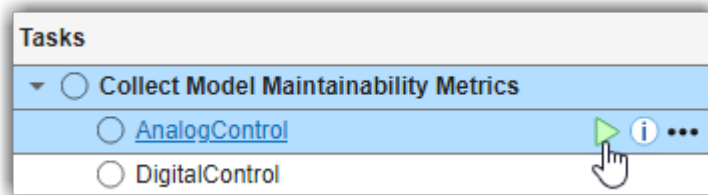


Each built-in task has a default behavior, but you can reconfigure the task to perform differently by specifying different property values for the task object in the process model. You can also create your own custom tasks with custom behaviors. For more information, see “Customize Your Process Model”.

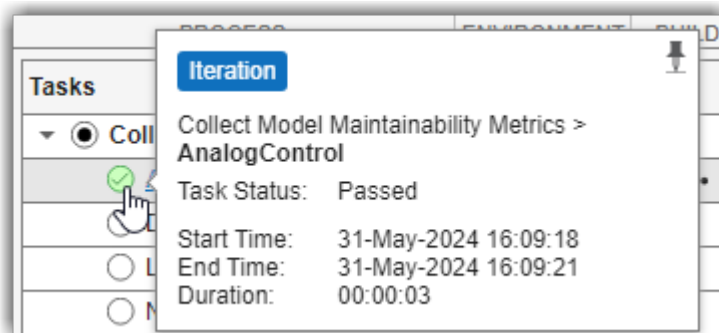
Run Tasks and Review Results

You can run tasks either from the Process Advisor app or by using the Process Advisor function `runprocess`.

- 1 To run a task on a specific artifact, you can point to that task iteration and click the run button. In the **Tasks** column, under **Collect Model Maintainability Metrics**, point to the model name **AnalogControl** and click the run button ▶.



The **Collect Model Maintainability Metrics** task runs on the current model. Process Advisor logs task activity in the MATLAB Command Window. When the task runs successfully, the status in the **Tasks** column shows a green circle with a check mark ✓. When you point to the status icon, you can view details about the status, including the task status and how long the task took to run.



- 2 If you point to the file icon 📄 in the **I/O** column, the pop-up shows hyperlinks to the outputs from the task, and inputs and dependencies for the task. In the **Details** column, you can see that the task successfully output a model maintainability report.

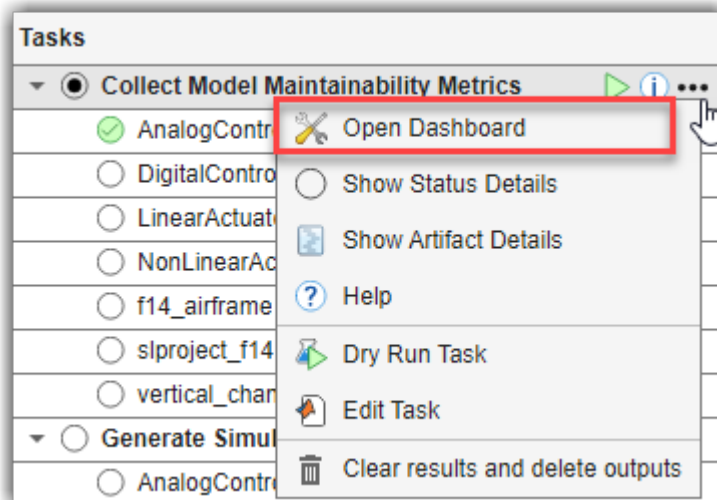
You can click the hyperlink to open and view the report.

	I/O	Details
		✓ 1
Outputs:		✓ 1
AnalogControl_ModelMaintainability.pdf		
Inputs:		
AnalogControl		
Dependencies:		
AnalogControl		
buses		
controller		
processmodel.m		

If you want the task to generate a different report format, such as HTML, or to collect only specific metrics, you can reconfigure the task to change the task behavior. For more information, see “Customize Your Process Model”.

- 3 To view the metric results directly in the Model Maintainability Dashboard, you can point to the task iteration, click ... > **Open Dashboard**.

The task automatically collects metric results that describe the size, complexity, and architecture of the model and those metric results appear in the dashboard.



- 4 To run a task on each of the artifacts, point to the task title and click the run button . Collect model maintainability metrics for each model in the project by clicking the run button for the task **Collect Model Maintainability Metrics**.

Process Advisor highlights and runs the necessary task and the task iterations. If the **Collect Model Maintainability Metrics** task depended on results from other tasks, Process Advisor would automatically run those tasks first.

Tasks	I/O	Details
<input checked="" type="radio"/> Collect Model Maintainability Metrics ▶ ⓘ ⋮		✓ 1
<input checked="" type="radio"/> AnalogControl		✓ 1
<input type="radio"/> DigitalControl		
<input type="radio"/> LinearActuator		
<input type="radio"/> NonLinearActuator		
<input type="radio"/> f14_airframe		
<input type="radio"/> slproject_f14		
<input type="radio"/> vertical_channel		

Run outdated tasks and dependent tasks

Process Advisor aggregates the results of each task. In the **Details** column, Process Advisor shows the number of passing, warning, or failing results:

- A green check mark ✓ indicates a passing result.
- An orange triangle ⚠ indicates a warning result.
- A red "X" ✗ indicates a failing result.

In this example, the **Collect Model Maintainability Metrics** task successfully collected the model maintainability metrics for 7 models, so the **Details** column shows a value of 7 next to the green check mark for the task.

The task options menu and Process Advisor toolstrip have additional options for running each of the tasks in your process, cleaning tasks to clear results and delete outputs, and other functionalities. For more information, see Process Advisor.

Identify Impact of Changes

If you make a change to an artifact in your project, Process Advisor can detect the change and automatically determine the impact of the change on your existing task results. For example, if you make a change to an artifact or its dependencies, certain task results are no longer up to date. Process Advisor can identify the impacted task results automatically and invalidate the task completion by marking the task results as outdated.

- 1 Open the AnalogControl model in the project by clicking the model name in the **Tasks** column. The artifact names shown in the **Tasks** column are hyperlinks to the artifacts.

Tasks	I/O	Details
▼ Collect Model Maintainability Metrics		✓ 7 3
AnalogControl ...		✓ 1
DigitalControl		✓ 1
LinearActuator		✓ 1
NonLinearActuator		✓ 1
f14_airframe		✓ 1
slproject_f14		✓ 1
vertical_channel		✓ 1 3

Note that a model-specific view of Process Advisor is available in a pane to the left of the Simulink canvas. You can use either view to interact with your tasks, but this example uses the standalone window instead. For more information, see Process Advisor.

- 2 In Simulink, make a change to the AnalogControl model and save the model. For this example, click and drag a block to a different part of the Simulink canvas, save, and close the model.

Process Advisor automatically detects the change to the project file and prompts you to refresh the tasks by using the **Refresh Tasks** button in the warning banner.

- 3 View the updated task statuses in the standalone Process Advisor window by clicking **Refresh Tasks**.

Process Advisor marks the task statuses for both the AnalogControl model and the slproject_f14 model as outdated.

Tasks	I/O	Details
▼ Collect Model Maintainability Metrics		✓ 7 3
AnalogControl		✓ 1
DigitalControl		✓ 1
LinearActuator		✓ 1
NonLinearActuator		✓ 1
f14_airframe		✓ 1
slproject_f14		✓ 1
vertical_channel		✓ 1 3

- 4 For the **slproject_f14** task iteration, point to the file icon in the **I/O** column.

Process Advisor shows that the app marked the task results for slproject_f14 model as outdated because the slproject_f14 references the AnalogControl model, which became invalidated.

- 2 In the MATLAB Command Window, you can find a summary of which tasks were run or skipped at the end of the log.

```
#####
## Ending Process Advisor build at 31-May-2024 17:45:05
#### Duration:                00:00:08
#### Build Status:            Pass
#### Number of tasks:         7
#### Number of tasks executed: 2
#### Number of tasks skipped: 5
#### Number of tasks in queue: 0
#### Number of tasks failed:  0
#### Tasks that were skipped:(Status::Task::IterationArtifact)
#### Pass::padv.builtin.task.CollectMetrics::models/DigitalControl.slx
#### Pass::padv.builtin.task.CollectMetrics::models/LinearActuator.slx
#### Pass::padv.builtin.task.CollectMetrics::models/NonLinearActuator.slx
#### Pass::padv.builtin.task.CollectMetrics::models/f14_airframe.slx
#### Pass::padv.builtin.task.CollectMetrics::models/vertical_channel.slx
#####
```

If you want Process Advisor to always force tasks to rerun, you can turn off incremental builds by clicking **Settings** in the toolstrip and clearing the **Incremental build** check box. For more information about settings, see “Specify Settings for Process Advisor and Build System” on page 1-16.

Export Build Report

You can export a build report that summarizes the Process Advisor task statuses, task results, and other information about the task execution.

In the Process Advisor toolstrip, in the **Export** section, click **Report**. In the Export Report dialog box, you can specify options for the report and export a report for the current process by clicking **Export**.

Alternatively, you can programmatically generate a report by using `padv.ProcessAdvisorReportGenerator` to specify options for the report and `generateReport` to generate the report. For example:

```
rptObj = padv.ProcessAdvisorReportGenerator;
generateReport(rptObj)
```

For more information, see “Generate Build Report” on page 1-14.

Explore Other Options

You can use Process Advisor to automate and run tasks on your machine and deploy a consistent development and verification process across your team. If you use source control and continuous integration (CI) for your project, you can also use Process Advisor as part of your prequalification process to make sure your team runs specific tasks before submitting their changes to source control. Having a consistent process, defined by the process model, can help your team prevent build and test failures in CI.

Use this table to find more information based on your goals.

Goal	Related Information
Learn more about the Process Advisor app.	Process Advisor
Customize the pipeline of tasks by reconfiguring the built-in tasks, removing tasks, and adding custom tasks.	"Customize Your Process Model"
Integrate into a continuous integration (CI) system.	"Integrate Process into CI"
Debug failures seen in CI.	"Locally Reproduce Issues Found in CI" on page 1-20

See Also

Process Advisor | generateReport | runprocess

Programmatically Run Tasks

With the support package CI/CD Automation for Simulink Check, you can run the tasks in your development and verification process by using the Process Advisor app on your local desktop or the Process Advisor function `runprocess`. Both of these approaches invoke the same incremental build system so that you can have consistent task execution across different environments like local desktop machines and continuous integration (CI) agents. The *build system* is software that can create the pipeline of tasks, efficiently execute tasks in the pipeline, and perform other actions related to the pipeline.

This example shows how you can run tasks and generate a build report for a project programmatically by using the `runprocess` function with other supporting functions. You can call these commands on your local desktop and from CI agents. For information on running tasks with Process Advisor, see Process Advisor.

Run Pipeline of Tasks

You can run tasks programmatically by using the `runprocess` function.

Run All Tasks

By default, if you use the `runprocess` function without specifying any name-value arguments, `runprocess()`, the function runs each of the tasks associated with the current project and process model. This behavior is equivalent to clicking the **Run All** button in the Process Advisor app.

To run each of the tasks associated with the current project and process, enter:

```
runprocess()
```

Run Specific Task

However, you often only want to run certain tasks or only run certain tasks on certain artifacts.

To only run a specific set of tasks, provide the task names to the `Tasks` argument. For example:

```
% run the Generate Simulink Web View task
% and the Check Modeling Standards tasks
runprocess(...
Tasks = ["padv.builtin.task.GenerateSimulinkWebView",...
"padv.builtin.task.RunModelStandards"])
```

Run Task Iterations for Specific Artifact

To only run the task iterations associated with a specific artifact, use the `FilterArtifact` argument. For example, to run tasks for the `AHRS_Voter` model, you can specify the value as the relative path to the model:

```
% run only the AHRS_Voter tasks
runprocess(...
FilterArtifact = fullfile(...
"02_Models", "AHRS_Voter", "specification", "AHRS_Voter.slx"))
```

For more information, see the function `runprocess`.

View Available Tasks Iterations

To return a list of the available task iterations in your current process, you can use the `generateProcessTasks` function.

```
generateProcessTasks
```

You can include or exclude certain task iterations by using the name-value arguments of `generateProcessTasks`. For example, to list the task iterations associated with a specific model, you can specify the relative path to the model using a `padv.Artifact` object and pass that object to the `FilterArtifact` argument for `generateProcessTasks`.

```
% specify the relative path to the model AHRS_Voter
model = padv.Artifact("sl_model_file",...
padv.util.ArtifactAddress(...
fullfile("02_Models","AHRS_Voter","specification","AHRS_Voter.slx"));

% find the tasks associated with the model AHRS_Voter
ahrsVoterTasks = generateProcessTasks(FilterArtifact=model)
```

Generate Build Report

You can generate a report that summarizes the build results for the tasks that you run in your pipeline.

The report includes a:

- Summary of task statuses
- Summary of task results
- Details about the task configuration and execution

Generate Report After Running Process

To automatically generate a report after you run your process, specify the `GenerateReport` argument of the `runprocess` function as `true`:

```
runprocess(GenerateReport = true)
```

By default, the report generates as a PDF file in the current working directory. You can use the `ReportFormat` and `ReportPath` arguments to specify a different report format and a different report name or full file path:

```
runprocess(GenerateReport = true,...
ReportFormat = "html-file",...
ReportPath = fullfile(pwd,"folderName","reportName"))
```

Generate Report from Recent Task Results

After you run the tasks in your pipeline, you can also generate a report using the most recent task results.

After you run a task, create a `padv.ProcessAdvisorReportGenerator` report object.

```
rptObj = padv.ProcessAdvisorReportGenerator;
```

Run `generateReport` on the report object to generate a build report in the current directory.

```
generateReport ( rptObj )
```

By default, the report generator generates a PDF. To generate an HTML report, specify the Format of the ProcessAdvisorReportGenerator object as `html - file`.

```
htmlReport=padv.ProcessAdvisorReportGenerator(Format="html-file");  
generateReport (htmlReport);
```

If you run the tasks in the default process model, the report provides an overview common development and verification activities like the:

- Model Advisor analysis, including the number of passing, warning, and failing checks
- Test results, organized by iteration
- Generated code files
- Coding standards checks

For information on how to get started with the default process model, see “Modify Default Process Model to Fit Your Process” on page 2-2.

See Also

`generateProcessTasks` | `padv.Artifact` | `padv.util.ArtifactAddress` | Process Advisor | `runprocess`

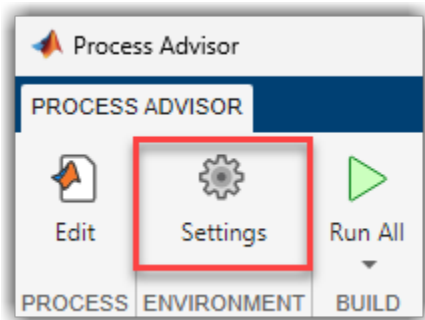
Related Examples

- “Automate and Run Tasks with Process Advisor” on page 1-2
- “Best Practices for Effective Builds” on page 3-32
- “Integrate Process into CI”
- “Specify Settings for Process Advisor and Build System” on page 1-16

Specify Settings for Process Advisor and Build System

With the CI/CD Automation for Simulink Check support package, you can run the tasks in your development and verification process by using Process Advisor and its incremental build system. There are several settings that you can use to customize how the Process Advisor app and `runprocess` function run tasks, cache information, and log results. For example, you can use settings to use incremental builds, enable model caching, and customize other behaviors.

You can access and change settings by clicking the **Settings** button in the Process Advisor toolstrip and selecting or clearing the check boxes for individual settings.





















There are two types of settings:

- “Project Settings” on page 1-16 — These settings are stored in the project and are shared with everyone using this project.
- “User Settings” on page 1-18 — These settings only apply to the current user.

Project Settings

These settings are stored in the project and are shared with everyone using this project. For additional settings and information, see `padv.ProjectSettings`.

Setting	Usage												
Incremental build	<p>Select this setting to allow the build system to automatically detect changes and mark task results as outdated.</p> <p>By default, the build system and the Process Advisor app perform incremental builds. Incremental builds can help you reduce the number of task iterations that you need to re-run by identifying and running only the task iterations with outdated results. If the task iteration results are up-to-date, the build system and the Process Advisor app skip the task iteration.</p> <p>When incremental builds are enabled, the task status icons in the Tasks column indicate whether the task results are up-to-date or outdated. Up-to-date task results have task status icons that are green for tasks that pass and red for tasks that fail or generate errors. Outdated task results have task status icons that are gray.</p> <div data-bbox="873 995 1435 1549" style="border: 1px solid gray; padding: 5px;"> <p>Tasks</p> <table border="1"> <tbody> <tr> <td style="text-align: center;"></td> <td>Task A (up-to-date)</td> </tr> <tr> <td style="text-align: center;"></td> <td>Task B (up-to-date)</td> </tr> <tr> <td style="text-align: center;"></td> <td>Task C (up-to-date)</td> </tr> <tr> <td style="text-align: center;"></td> <td>Task D (outdated)</td> </tr> <tr> <td style="text-align: center;"></td> <td>Task E (outdated)</td> </tr> <tr> <td style="text-align: center;"></td> <td>Task F (outdated)</td> </tr> </tbody> </table> </div> <p>Default: On</p>		Task A (up-to-date)		Task B (up-to-date)		Task C (up-to-date)		Task D (outdated)		Task E (outdated)		Task F (outdated)
	Task A (up-to-date)												
	Task B (up-to-date)												
	Task C (up-to-date)												
	Task D (outdated)												
	Task E (outdated)												
	Task F (outdated)												
Enable model caching	<p>Select this setting to allow the build system to cache models during builds.</p> <p>Default: Off</p>												

Setting	Usage
Suppress outputs to command window	<p>Select this setting to suppress the build log and task execution messages in the MATLAB Command Window. This setting only applies when MATLAB is in interactive mode, not batch mode.</p> <p>Default: Off</p>
Show file extensions	<p>Select this setting to show file extensions for all task iteration artifacts in the Tasks column in Process Advisor.</p> <p>To keep file extensions in the results for a specific query, you can specify the query property <code>ShowFileExtension</code> as <code>true</code>. For information, see <code>padv.Query</code>.</p> <p>Default: Off</p>
Untracked dependency behavior	<p>Build system behavior when there are untracked I/O files, specified as either:</p> <ul style="list-style-type: none"> • Allow — Do not generate warnings or errors for untracked I/O files. • Warn — Generate a warning if a task has untracked I/O files. • Error — Generate an error if a task has untracked I/O files. <p>If you make a change to an untracked file, Process Advisor and the build system <i>do not</i> mark the task as outdated. For considerations and best practices, see “Review Untracked Dependencies” on page 2-56. For more information on untracked files and change tracking, see “Exclude Files from Change Tracking in Process Advisor” on page 2-59.</p> <p>Default: Warn</p>

User Settings

These settings only apply to the current user. For additional settings and information, see `padv.UserSettings`.

Setting	Usage
Detect duplicate outputs	<p>Select this setting to allow the build system to generate an error message when multiple tasks attempt to write to the same output file.</p> <p>Default: On</p>

Setting	Usage
Garbage collect task outputs	<p>Select this setting to allow the build system to automatically clean task results for tasks and artifacts that do not match the current process model or project.</p> <p>Default: On</p>
Show detailed error messages	<p>Select this setting to allow the build system to show more information in error messages. By default, error messages from the build system are not verbose.</p> <p>Default: Off</p>
Add process model as dependency	<p>Select this setting to add the process model file as a dependency.</p> <p>By default, if you make a change to the process model file, the build system marks each task status and task result as outdated because the tasks in the updated process model might not match the tasks that generated the task results from the previous version of the process model.</p> <p>If you do not want changes to the process model to make task statuses and task results outdated, clear this setting.</p> <p>For more information on untracked files and change tracking, see “Exclude Files from Change Tracking in Process Advisor” on page 2-59.</p> <p>Default: On</p>

See Also

`padv.ProjectSettings` | `padv.UserSettings` | Process Advisor | `runprocess`

Related Examples

- “Automate and Run Tasks with Process Advisor” on page 1-2
- “Best Practices for Process Model Authoring” on page 2-56
- “Best Practices for Effective Builds” on page 3-32

Locally Reproduce Issues Found in CI

With the CI/CD Automation for Simulink Check support package, you can run your process in CI, download the job artifacts, and locally view the results in Process Advisor. If there were failures in CI, you can use Process Advisor to debug and find issues in your artifacts that you need to fix on your local machine. You can copy results from CI jobs onto your local machine by cloning a copy of the project that you ran in CI and copying the latest job artifacts.

For information about how to run your process in CI, see “Integrate Process into CI”.

Get Latest Project Files

- 1 In MATLAB, get the latest project files by cloning a copy of the project onto your local machine. For more information, see “Clone Git Repository in MATLAB”.
- 2 Close your local copy of the project. You must close the project before you attempt to copy CI artifacts into the project folder.

Download and Copy CI Artifacts into Project

- 1 In your CI system, open the job that you want to inspect locally and download the artifacts that the job generated. Job artifacts typically download as a ZIP file.

If you are using the pipeline generator, `padv.pipeline.generatePipeline`, the **Collect_Artifacts** job automatically collects and compresses the build artifacts from your pipeline into a ZIP file that you can download.

- 2 Close your local copy of the project if you have it open in MATLAB.
- 3 Extract the files from the ZIP file and copy the artifacts into the folder for your local copy of the project. The copied artifacts do not need to be added to the MATLAB path or project path.
- 4 Open your local copy of the project in MATLAB.
- 5 Open the Process Advisor app. If you see a warning banner, click **Refresh Tasks**.

Debug in Process Advisor

After you refresh the tasks, you can use Process Advisor to:

- See the task results from the CI job in your local Process Advisor app
- Re-run tasks locally to reproduce the CI failure on your local machine
- Make changes to your project to fix the issues observed in CI
- Re-run tasks locally to confirm that you resolve open issues before submitting to source control

For more information, see “Automate and Run Tasks with Process Advisor” on page 1-2

Considerations for Parallel Code Generation

Starting in R2023b Update 5, the pipeline generator supports a round-trip, parallel CI workflow that automatically merges the task statuses and project analysis from across the parallel branches. For information, see “Parallel Pipeline Architectures” on page 3-24.

If you are using the pipeline generator in a previous release and you specify a parallel pipeline architecture like `IndependentModelPipelines`, each parallel pipeline generates separate artifact

database files, `artifacts.dmr`, for each parallel branch. The build system and Process Advisor app can only load one `artifacts.dmr` file at a time, so if you try to view the generated task statuses and results on your local machine, you see incomplete or outdated task statuses.

See Also

Related Examples

- “Approaches to Running Processes in CI” on page 3-2
- “Clone Git Repository in MATLAB”
- “Integrate Process into GitHub” on page 3-5
- “Integrate Process into GitLab” on page 3-8
- “Integrate Process into Jenkins” on page 3-14
- “Integrate Process into Other CI Platforms” on page 3-19

Customize Your Process Model

Modify Default Process Model to Fit Your Process

With the CI/CD Automation for Simulink Check support package, you can define a consistent process for your team by using a process model file. When your team has a standard process for local prequalification and CI builds, you can efficiently enforce guidelines and make collaboration easier. This example shows how to reconfigure the default process model to create a consistent, repeatable process that you can deploy to your team. In this example, you take the default process model and modify the tasks and queries to fit your requirements.

For more information about the process model, see “Overview of Process Model” on page 2-9.

Open Project

Open a project that contains your files. If you do not already have a project, you can create a project as shown in “Create Projects”.

Create Process Model

You can create a process model for your project by using either the:

- Process Advisor app — When you open Process Advisor on a project that does not have a process model, the app automatically copies the default process model into the project.
- `createprocess` function — You can use this function to access the different process model templates, including a template for the default process model.

Inspect Default Process Model

Inspect the default process model by opening the Process Advisor app and clicking the **Edit** button .

The default process model has four main sections that you can edit to fit your development and verification workflow:

- “Section A — Add or Remove Built-In Tasks” on page 2-4
- “Section B — Modify Behavior of Built-In Tasks” on page 2-6
- “Section C — Specify Dependencies Between Tasks” on page 2-7
- “Section D — Specify Preferred Task Execution Order” on page 2-7

In the following diagram, the letters A, B, C, and D indicate the location of those sections in the default process model.


```

1 function processmodel(pm)
2     % Defines the project's processmodel
3
4     arguments
5         pm padv.ProcessModel
6     end
7
8     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
9     %% Include/Exclude Tasks in processmodel
10    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
11
12    includeModelMaintainabilityMetricTask = true;
13    includeModelTestingMetricTask = true;
14    includeModelStandardsTask = true;
15    includeDesignErrorDetectionTask = false;
16    includeModelComparisonTask = false;
17    includeSDDTask = true;
18    includeSimulinkWebViewTask = true;
19    includeTestsPerTestCaseTask = true;
20    includeMergeTestResultsTask = true;
21    includeGenerateCodeTask = true;
22    includeAnalyzeModelCode = true && ~padv.internal.isMACA64 && exist('polyspaceroot','file');
23    includeProveCodeQuality = true && ~padv.internal.isMACA64 && (~isempty(ver('pscodeprover')) || ~isempty(ver('pscodeproverserver')));
24    includeCodeInspection = false;
25
26    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
27    %% Define Shared Path Variables
28    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29
30    % Set default root directory for task results
31    pm.DefaultOutputDirectory = fullfile('$PROJECTROOT$', 'PA_Results');
32    defaultResultPath = fullfile( ...
33        '$DEFAULTOUTPUTDIR$', '$ITERATIONARTIFACTS$');
34
35    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
36    %% Define Shared Queries
37    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
38    findModels = padv.builtin.query.FindModels(Name="ModelsQuery");
39    findModelsWithTests = padv.builtin.query.FindModelsWithTestCases(Parent=findModels);
40    findTestsForModel = padv.builtin.query.FindTestCasesForModel(Parent=findModels);
41
42    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
43    %% Register Tasks
44    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
45
46    %% Collect Model Maintainability Metrics
47    % Tools required: Simulink Check
48    if includeModelMaintainabilityMetricTask
49        mmMetricTask = pm.addTask(padv.builtin.task.CollectMetrics());
50    end

```

```

157 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
158 %% Set Task relationships
159 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
160
161 %% Set Task Dependencies
162 C if includeGenerateCodeTask && includeCodeInspection
163     slciTask.dependsOn(codegenTask);
164 end
165 if includeGenerateCodeTask && includeAnalyzeModelCode
166     psbfTask.dependsOn(codegenTask);
167 end
168 if includeGenerateCodeTask && includeProveCodeQuality
169     pscpTask.dependsOn(codegenTask);
170 end
171 if includeTestsPerTestCaseTask && includeMergeTestResultsTask
172     mergeTestTask.dependsOn(milTask, "WhenStatus", {'Pass', 'Fail'});
173 end
174
175 %% Set Task Run-Order
176 D if includeMergeTestResultsTask && includeModelTestingMetricTask
177     mtMetricTask.runsAfter(mergeTestTask);
178 end
179 if includeSimulinkWebViewTask && includeModelMaintainabilityMetricTask
180     slwebTask.runsAfter(mmMetricTask);
181 end
182 if includeModelStandardsTask && includeModelMaintainabilityMetricTask
183     maTask.runsAfter(mmMetricTask);
184 end
185 if includeModelStandardsTask && includeSimulinkWebViewTask
186     maTask.runsAfter(slwebTask);

```

Section A — Add or Remove Built-In Tasks

The default process model adds several built-in tasks to the process, including tasks for collecting metrics and running checks with Model Advisor. You can add or remove these built-in tasks from your process by setting the variable associated with a built-in task to `true` or `false` in your process model. For example, to have your process include the built-in task for generating Simulink model comparisons, edit the process model to set `includeModelComparisonTask` to `true`.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Include/Exclude Tasks in processmodel
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

includeModelMaintainabilityMetricTask = true;
includeModelTestingMetricTask = true;
includeModelStandardsTask = true;
includeDesignErrorDetectionTask = false;
includeFindClones = true;
includeModelComparisonTask = true;
includeSDDTask = true;
includeSimulinkWebViewTask = true;
includeTestsPerTestCaseTask = true;
includeMergeTestResultsTask = true;
includeGenerateCodeTask = true;
includeAnalyzeModelCode = true && exist('polyspaceroot', 'file');

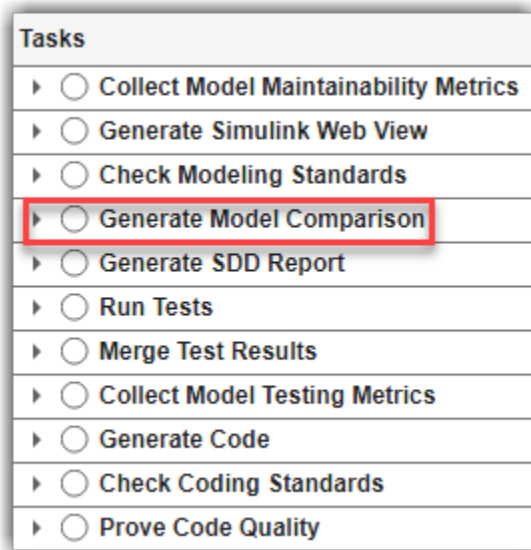
```

```
includeProveCodeQuality = true && (~isempty(ver('pscodeprover')) || ~isempty(ver('pscodeproverse
includeCodeInspection = false;
```

Each variable is associated with a task in the default process model.

Variable in Default Process Model	Task Title	Built-In Task
includeModelMaintainabilityMetricTask	Collect Model Maintainability Metrics	padv.builtin.task.CollectMetrics
includeModelTestingMetricTask	Collect Model Testing Metrics	padv.builtin.task.CollectMetrics
includeModelStandardsTask	Check Modeling Standards	padv.builtin.task.RunModelStandards
includeDesignErrorDetectionTask	Detect Design Errors	padv.builtin.task.DetectDesignErrors
includeFindClones	Find Clones	padv.builtin.task.FindClones
includeModelComparisonTask	Generate Model Comparison	padv.builtin.task.GenerateModelComparison
includeSDDTask	Generate SDD Report	padv.builtin.task.GenerateSDDReport
includeSimulinkWebViewTask	Generate Simulink Web View	padv.builtin.task.GenerateSimulinkWebView
includeTestsPerTestCaseTask	Run Tests	padv.builtin.task.RunTestsPerTestCase
includeMergeTestResultsTask	Merge Test Results	padv.builtin.task.MergeTestResults
includeGenerateCodeTask	Generate Code	padv.builtin.task.GenerateCode
includeAnalyzeModelCode	Check Coding Standards	padv.builtin.task.AnalyzeModelCode
includeProveCodeQuality	Prove Code Quality	padv.builtin.task.AnalyzeModelCode
includeCodeInspection	Inspect Code	padv.builtin.task.RunCodeInspection

The tasks that you add in the process model appear in the **Tasks** column in Process Advisor. In addition to the built-in tasks, you can also add custom tasks to your process model. For more information, see “Add Tasks to Process” on page 2-13.



Section B – Modify Behavior of Built-In Tasks

The built-in tasks have default behaviors, but you can modify how a task performs its action by reconfiguring the task object in the process model. For example, the built-in task `padv.builtin.task.RunModelStandards` has a property `ReportPath` that specifies where the task saves the output Model Advisor report. By default, the `RunModelStandards` task saves the report in a subfolder named `model_standards`. The default process model reconfigures the task object, `maTask`, to have the task save the Model Advisor report in a subfolder named `model_standards_results` instead. You can modify other task behaviors by setting other task object property values. For example, to have the task generate the Model Advisor report as a Microsoft® Word document instead of HTML, set `ReportFormat` to "docx".

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Register Tasks
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Checking model standards on a model
if includeModelStandardsTask
    maTask = pm.addTask(padv.builtin.task.RunModelStandards());
    maTask.ReportPath = fullfile( ...
        defaultResultPath, 'model_standards_results');
    maTask.ReportFormat = "docx";
end

...

```

When you run the task in Process Advisor, the task performs its action using the behaviors that you specified in the process model. You can view the results in the **I/O** column of Process Advisor. To see other examples of how you can reconfigure tasks, inspect the default process model and see “Reconfigure Task Behavior” on page 2-17.

Tasks	I/O	Details
▶ <input type="radio"/> Collect Model Maintainability Metrics		
▶ <input type="radio"/> Generate Simulink Web View		
▶ <input checked="" type="radio"/> Check Modeling Standards		✓ 99+ ▲ 15
▶ <input type="radio"/> Generate Model Comparison		
▶ <input type="radio"/> Generate SDD Report		
▶ <input type="radio"/> Run Tests		
▶ <input type="radio"/> Merge Test Results		
▶ <input type="radio"/> Collect Model Testing Metrics		
▶ <input type="radio"/> Generate Code		

Outputs:

- Actuator_Control_ModelAdvisor.docx
- AHRS_Voter_ModelAdvisor.docx
- Flight_Control_ModelAdvisor.docx
- InnerLoop_Control_ModelAdvisor.docx
- OuterLoop_Control_ModelAdvisor.docx

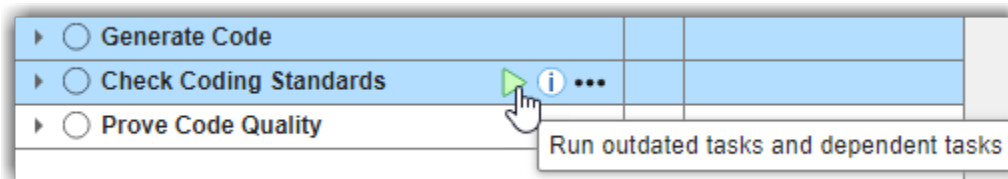
Section C — Specify Dependencies Between Tasks

Typically in your development and verification workflow, you have tasks that need inputs or results from other tasks in order to run successfully. In the process model, you specify these dependencies by using the `dependsOn` method on the task object.

For example, the default process model adds the built-in tasks **Generate Code** and **Check Coding Standards** to the process. Since you need to generate the code before you can analyze it, the default process model specifies that if your process model contains both the code generation and code analysis tasks, then the code analysis task object, `psbfTask`, needs to depend on the code generation task object `codegenTask`.

```
%% Set Task Dependencies
if includeGenerateCodeTask && includeAnalyzeModelCode
    psbfTask.dependsOn(codegenTask);
end
```

If you open Process Advisor and point to the run button for the **Check Coding Standards** task, Process Advisor highlights dependency on the **Generate Code** task. If you try to run the **Check Coding Standards** task, the build system automatically runs the **Generate Code** task first. For more information, see “Define Task Relationships” on page 2-21.



Section D — Specify Preferred Task Execution Order

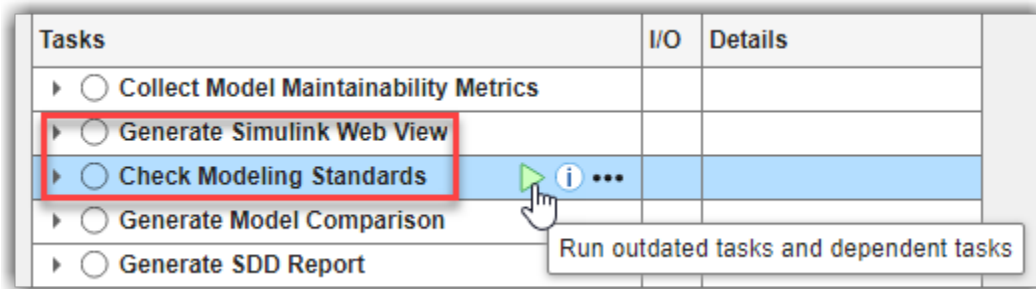
Often there are tasks in your development and verification workflow that you want to run in a specific order, even though the tasks do not depend on each other. To specify a preferred task execution order for tasks that do not depend on each other, you can use the `runsAfter` method on the task object.

For example, the default process model specifies that the modeling standards task (`maTask`) should run after the Simulink web view task (`slwebTask`). The modeling standards task does not depend on

information from the Simulink web view task in order to run, but that is the preferred execution order for the tasks in this particular process.

```
%% Set Task Run-Order
if includeModelStandardsTask && includeSimulinkWebViewTask
    maTask.runsAfter(slwebTask);
end
```

In Process Advisor, the **Check Modeling Standards** task appears after the **Generate Simulink Web View** task in the **Tasks** column. For more information on task ordering, see “Define Task Relationships” on page 2-21.



See Also

padv.ProcessModel

Related Examples

- “Automate and Run Tasks with Process Advisor” on page 1-2
- “Overview of Process Model” on page 2-9

Overview of Process Model

You can define a repeatable development and verification process for your team by using the support package CI/CD Automation for Simulink Check. You define your process inside a process model. A process model is a MATLAB file that specifies the tasks that you want to perform and dependencies between those tasks. The support package has built-in tasks that you can add to your process to perform common activities like running Model Advisor checks, generating code, and running tests. But you can also create and add your own custom tasks to your process. To specify which artifacts your tasks iterate over or use as task inputs, you can use built-in and custom queries to automatically find certain types of artifacts.

To get started with the default process model, see “Modify Default Process Model to Fit Your Process” on page 2-2.

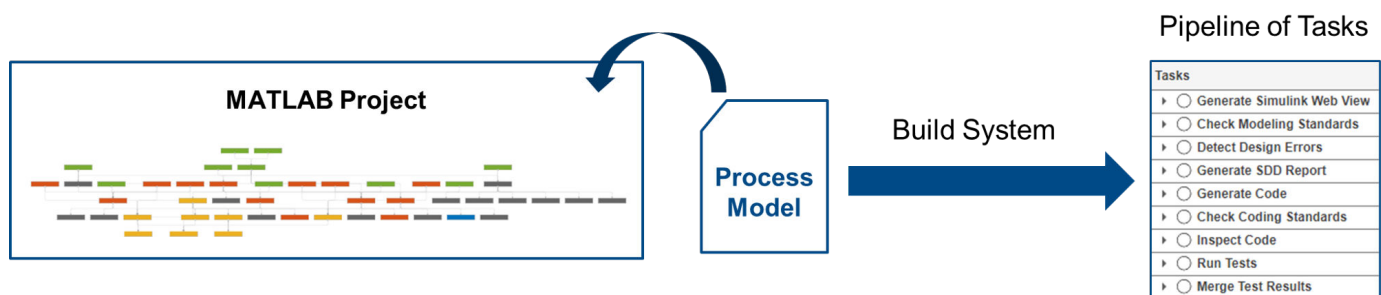
Process Model

You define your process by using a process model file. A process model file accepts one argument, a `padv.ProcessModel` object. You define your process by modifying the `padv.ProcessModel` object and related objects.

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end
end
```

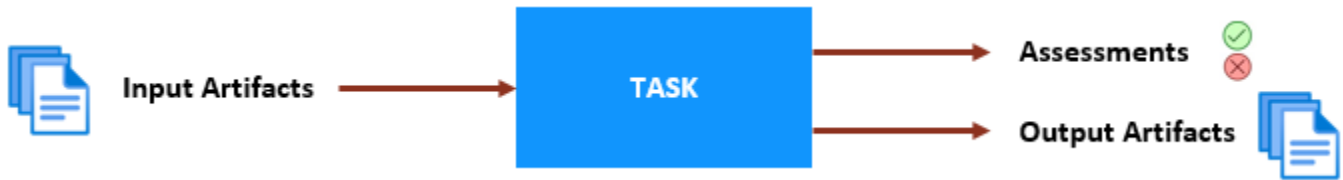
The process model file for your project must:

- Be on the MATLAB path.
- Have the file name `processmodel.m` or `processmodel.p`.



Tasks

In the process model, you add the tasks that you want to perform as part of your process. A task represents an individual step in the process. Tasks can take artifacts as inputs, perform specific actions, generate assessments, and return artifacts as outputs.



You add tasks to your process by using the `addTask` object function on the `padv.ProcessModel` object.

You can add:

- “Built-In Tasks” on page 2-14 for common activities like checking modeling standards, generating code, and running tests. The classes that define the built-in tasks are in the `padv.builtin.task` namespace.
- “Custom Tasks” on page 2-15 for your own customized task behavior. Your custom tasks can inherit from one of the built-in task classes or the `padv.Task` superclass.

For example, to add the built-in task for checking modeling standards with Model Advisor, you pass the task name or task instance to the `addTask` object function. `addTask` returns a task object that you can use to reconfigure the task behavior for the current process.

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    % Add Task
    modelAdvisorTask = pm.addTask(padv.builtin.task.RunModelStandards);
end
```

In Process Advisor, the **Tasks** column shows the task and the artifacts that the task iterates over. By default, the built-in task `RunModelStandards` runs once for each model in the project. But you can reconfigure how often the task runs and other task behaviors.

Tasks	I/O	Details
<input type="checkbox"/> Check Modeling Standards		
<input type="checkbox"/> AHRS_Voter		
<input type="checkbox"/> Actuator_Control		
<input type="checkbox"/> Flight_Control		
<input type="checkbox"/> InnerLoop_Control		
<input type="checkbox"/> OuterLoop_Control		

For more information, see:

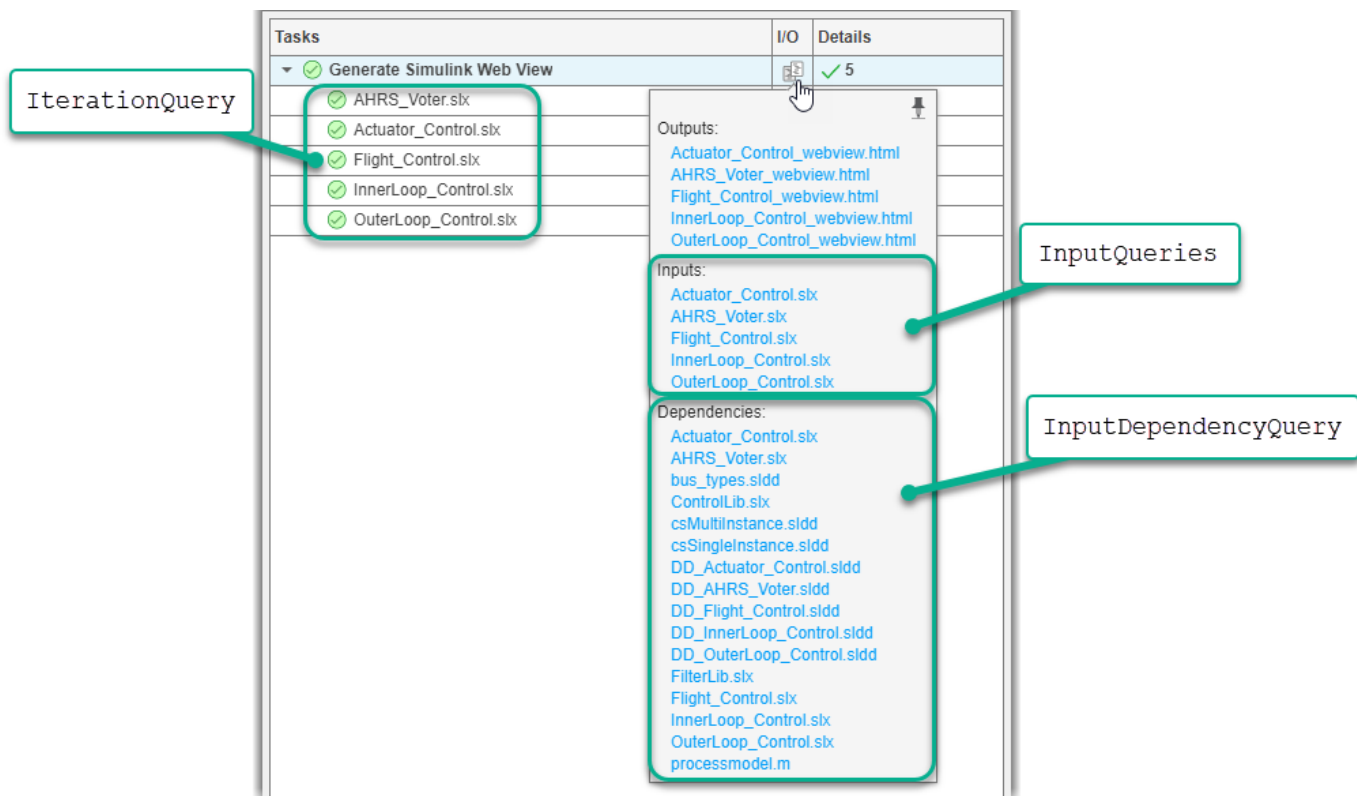
- “Add Tasks to Process” on page 2-13
- “Reconfigure Task Behavior” on page 2-17

- “Define Task Relationships” on page 2-21

Queries

You can find and use artifacts in your project by using queries. A query can automatically find artifacts in your project based on search criteria like the artifact type, project label, file path, and other characteristics. You can use queries in your process model to automatically find the relevant artifacts for your tasks without needing to manually update a static list of files.

In Process Advisor, the **Tasks** column shows the artifacts that a task iterates over. When you point to task results in the **I/O** column, you can see the task inputs and input dependencies.



You can reconfigure a task to iterate over different artifacts or use different task inputs by specifying a different query for the associated task property:

- **IterationQuery** — Determines how often a task runs by finding the artifacts that the task iterates over
- **InputQueries** — Finds inputs for the task
- **InputDependencyQuery** — Finds additional dependencies related to the inputs

For each task in your process, the build system runs the task **IterationQuery** to determine which artifacts the task iterates over. The build system then creates a task iteration, runs any additional queries the task needs, runs the task, and saves the task results. Although tasks iterate over artifacts, tasks do not automatically use those artifacts as inputs unless those artifacts are specified by the task input queries. For each task iteration, the build system runs the **InputQueries** to find the inputs for that specific task iteration. For each input, the build system runs the **InputDependencyQuery** to

find additional dependencies that can impact if task results are up-to-date. The task automatically becomes outdated if you make a change to any of the task inputs or input dependencies.

You can specify these task properties and other process modeling properties by using:

- “Built-In Queries” on page 2-23 to find artifacts like models, test cases, and requirements. The classes that define the built-in queries are in the `padv.builtin.query` namespace.
- “Custom Queries” on page 2-25 to find artifacts that are not covered by the built-in queries.

For information on how to reconfigure tasks to use different artifacts, see “Reconfigure Task Behavior” on page 2-17.

Use Your Process

When you are ready to run your process, you do not manually run the process model file. Instead, you use the Process Advisor app or the `runprocess` function. The Process Advisor build system automatically loads your process model, analyzes your project, and creates your pipeline of tasks. For more information, see “Automate and Run Tasks with Process Advisor” on page 1-2 and “Programmatically Run Tasks” on page 1-13.

See Also

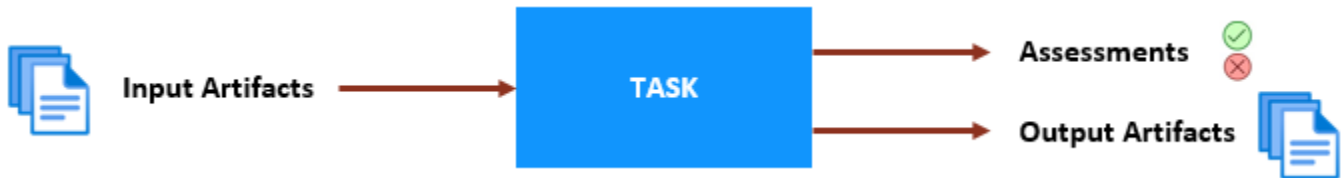
Process Advisor | `runprocess`

Related Examples

- “Automate and Run Tasks with Process Advisor” on page 1-2
- “Modify Default Process Model to Fit Your Process” on page 2-2
- “Programmatically Run Tasks” on page 1-13


Add Tasks to Process

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by creating a process model and adding tasks for the steps that you want to perform as part of your process. In the process model, you add the tasks that you want to perform as part of your process. A task represents an individual step in the process. Tasks can take artifacts as inputs, perform specific actions, generate assessments, and return artifacts as outputs.



Open Process Model

You can add tasks to your process by editing the process model file for your project. If you do not have a project or process model, see “Automate and Run Tasks with Process Advisor” on page 1-2 to get started.

- 1 Open the project that contains your files.
- 2 Open Process Advisor. On the **Project** tab, in the **Tools** section, click **Process Advisor**.
- 3 Edit the process model by clicking the **Edit** button  in the toolstrip.

Add Tasks

You add tasks to your process model by using the `addTask` object function on the `padv.ProcessModel` object. You can add:

- “Built-In Tasks” on page 2-14 for common activities like checking modeling standards, generating code, and running tests.
- “Custom Tasks” on page 2-15 for your own customized task behavior.

For example, the following process model adds the built-in task `padv.builtin.task.RunModelStandards` to the default process.

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    % Adding a built-in task
    modelAdvisorTask = pm.addTask(padv.builtin.task.RunModelStandards);

end
```

You can add multiple tasks to your process model, including multiple instances of the same task. However, each task object in your process must have a unique name specified by the `Name` property.

By default, tasks perform an action with a default behavior, but you can reconfigure the task behavior from inside the process model to change configuration files the task uses, output report file types, and other behaviors. For more information, see “Reconfigure Task Behavior” on page 2-17.

In Process Advisor, the **Tasks** column shows the task and the artifacts that the task iterates over.

Tasks	I/O	Details
▼ <input type="radio"/> Check Modeling Standards		
<input type="radio"/> AHRS_Voter		
<input type="radio"/> Actuator_Control		
<input type="radio"/> Flight_Control		
<input type="radio"/> InnerLoop_Control		
<input type="radio"/> OuterLoop_Control		

Built-In Tasks

The support package has built-in tasks for common activities like checking modeling standards, generating code, and running tests. The classes that define the built-in tasks are in the `padv.builtin.task` namespace. To view the source code for a built-in task, use the `open` function.

Goal	Task Title	Built-In Task	Required Product	Requires Display
Model Reports	Generate SDD Report	<code>padv.builtin.task.GenerateSDDReport</code>	Simulink Report Generator™	Yes. For more information, see “Set Up Virtual Display Machines Without Displays” on page 3-29.
	Generate Simulink Web View	<code>padv.builtin.task.GenerateSimulinkWebView</code>		
	Generate Model Comparison	<code>padv.builtin.task.GenerateModelComparison</code>	Simulink	
Model Analysis	Check Modeling Standards	<code>padv.builtin.task.RunModelStandards</code>	Simulink Check	No
	Detect Design Errors	<code>padv.builtin.task.DetectDesignErrors</code>	Simulink Design Verifier™	
Testing and Coverage	Merge Test Results	<code>padv.builtin.task.MergeTestResults</code>	Simulink Test	
	Run Tests	<code>padv.builtin.task.RunTestsPerModel</code>		

Goal	Task Title	Built-In Task	Required Product	Requires Display
	Run Tests	<code>padv.builtin.task.RunTestsPerTestCase</code>		
Model Design and Testing Metrics	Collect Metrics	<code>padv.builtin.task.CollectMetrics</code>	Simulink Check	
Code Generation	Generate Code	<code>padv.builtin.task.GenerateCode</code>	Embedded Coder	
Code Analysis	Check Coding Standards or Prove Code Quality	<code>padv.builtin.task.AnalyzeModelCode</code>	Polyspace® Bug Finder™ or Polyspace Code Prover™	
	Inspect Code	<code>padv.builtin.task.RunCodeInspection</code>	Simulink Code Inspector™	

Custom Tasks

If you need to perform steps that are not covered by the built-in tasks, you can create and add custom tasks to your process model.

For example, consider the following process model that adds a custom task named "RunMyScript" to run a script, `myScript.m`, and generate a result for Process Advisor. You define the action that the custom task performs by using the `Action` argument for the `addTask` method.

```
function processmodel(pm)

    arguments
        pm padv.ProcessModel
    end

    % Add custom task
    pm.addTask("RunMyScript", Action = @runMyScript);

end

% Define action that custom task performs
function taskResult = runMyScript(~)
    run("myScript.m");
    taskResult = padv.TaskResult;
end
```

However, for more complex tasks, you want to define your custom task in a separate class that inherits from one of the built-in task classes or from the `padv.Task` superclass. For more information, see "Create Custom Tasks" on page 2-28.

See Also

`addTask`

Related Examples

- “Overview of Process Model” on page 2-9
- “Reconfigure Task Behavior” on page 2-17

Reconfigure Task Behavior

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by adding tasks to your process model and reconfiguring the task behavior to meet the needs of your specific process.


You can modify the behavior of a task by overriding the values of the task properties in the process model. You can use the task properties to control the:

- “Task Inputs” on page 2-17
- “Task Action” on page 2-18
- “Task Iterations” on page 2-19

If you do not have a process model for your team, you can get started by using the default process model as shown in “Modify Default Process Model to Fit Your Process” on page 2-2.

Open Process Model

You can reconfigure task behavior for your project and process by editing the process model file for the project. If you do not have a project or process model, see “Automate and Run Tasks with Process Advisor” on page 1-2 to get started.

- 1 Open the project that contains your files.
- 2 Open Process Advisor. On the **Project** tab, in the **Tools** section, click **Process Advisor**.
- 3 Edit the process model by clicking the **Edit** button  in the toolbar.

Alternatively, for custom tasks, you can specify the task property values directly in your class definition file. For more information, see “Create Custom Tasks” on page 2-28.

Task Inputs

The `InputQueries` property of a task defines the task inputs. If you want to provide additional inputs to a task, you can add queries to the `InputQueries` property of the task by using the `addInputQueries` method on the task object in the process model. For each task in the process, Process Advisor runs the `InputQueries` property of the task to find the input artifacts. For each input artifact, Process Advisor also runs the `InputDependencyQuery` property of the task to find additional dependencies that can impact whether task results are up-to-date. Queries find artifacts in your project based on search criteria like the artifact type, project label, file path, and other characteristics. For more information, see “Find Artifacts with Queries” on page 2-23.

For example, if you want the **Check Modeling Standards** task to run the Model Advisor checks specified by a Model Advisor configuration file, `sampleChecks.json`, you can add this file as an input to the task. This allows the task to use the file as an input, recognize changes to the file, and update the task status if the file changes. The task automatically becomes outdated if you make a change to any of the task inputs or input dependencies.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
% Specify which Model Advisor configuration file to run
maTask.addInputQueries(padv.builtin.query.FindFileWithAddress( ...
    Type = 'ma_config_file',...
    Path = fullfile('tools','sampleChecks.json')));
```

When you run the task in Process Advisor and point to the task results in the **I/O** column, you can see the task **Inputs** and the additional input **Dependencies**.

Tasks	I/O	Details
▶ ○ Collect Model Maintainability Metrics		
▶ ○ Generate Simulink Web View		
▼ ✓ Check Modeling Standards	✓ 17 △ 3	
✓ AHRs_Voter	✓ 3 △ 1	
✓ Actuator_Control		
✓ Flight_Control		
✓ InnerLoop_Control		
✓ OuterLoop_Control		
▶ ○ Generate Model Comparison		
▶ ○ Generate SDD Report		
▶ ○ Run Tests		
▶ ○ Merge Test Results		
▶ ○ Collect Model Testing Metrics		
▶ ○ Generate Code		
▶ ○ Check Coding Standards		
▶ ○ Prove Code Quality		

Outputs:
AHRs_Voter_ModelAdvisor.html

Inputs:
AHRs_Voter.slx
sampleChecks.json

Dependencies:
AHRs_Voter.slx
bus_types.sldd
csSingleInstance.sldd
DD_AHRs_Voter.sldd
processmodel.m

Using Task Outputs as Task Inputs

If you want a task to use the outputs from a previous task:

- Specify a `dependsOn` relationship between the two tasks
- Update the `InputQueries` property of the downstream task to use the query `padv.builtin.query.GetOutputsOfDependentTask` as one of the input queries

For example, the built-in task `MergeTestResults` requires outputs from the built-in task `RunTestsPerTestCase`. In the process model, you must specify a dependency between these tasks by using the `dependsOn` function. For example:

```
mergeTestTask.dependsOn(milTask, "WhenStatus", {'Pass', 'Fail'});
```

If you open the source code for the built-in task `MergeTestResults`, you can see that the task uses the built-in query `GetOutputsOfDependentTask` as an input query to find the outputs from the `RunTestsPerTestCase` task.

```
...
options.InputQueries = [padv.builtin.query.GetIterationArtifact,...
    padv.builtin.query.GetOutputsOfDependentTask(...
        Task="padv.builtin.task.RunTestsPerTestCase")];
...
```

Task Action

Tasks have various properties that determine how they perform their task actions. For example, the **Check Modeling Standards** task has properties such as `CheckIDList`, `DisplayResults`, and

ExtensiveAnalysis. When you run the `RunModelStandards` task, these properties specify the input arguments for the function `ModelAdvisor.run`. For information on the built-in task classes and their properties, see “Built-In Tasks” on page 2-14.

You can reconfigure how a task runs by specifying different values for these properties in the process model. For example, for the **Check Modeling Standards** task, you can specify a list of Model Advisor checks to run and the report format by setting the `CheckIDList` and `ReportFormat` properties.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
% Specify which Model Advisor checks to run
maTask.CheckIDList = {'mathworks.jmaab.db_0032', 'mathworks.jmaab.jc_0281'};
% Specify report format
maTask.ReportFormat = 'docx';
```

When you run the task in Process Advisor, the task runs the specified Model Advisor checks and generates the Model Advisor report as a Microsoft Word document instead of an HTML file.

Tasks	I/O	Details
▶ ○ Collect Model Maintainability Metrics		
▶ ○ Generate Simulink Web View		
▼ ✓ Check Modeling Standards	✓ 8 △ 2	
✓ AHRs_Voter	✓ 1 △ 1	
✓ Actuator_Control		
✓ Flight_Control		
✓ InnerLoop_Control		
✓ OuterLoop_Control		
▶ ○ Generate Model Comparison		
▶ ○ Generate SDD Report		
▶ ○ Run Tests		
▶ ○ Merge Test Results		
▶ ○ Collect Model Testing Metrics		
▶ ○ Generate Code		

Outputs: AHRs_Voter_ModelAdvisor.docx
Inputs: AHRs_Voter.slx
Dependencies: AHRs_Voter.slx bus_types.sldd csSingleInstance.sldd DD_AHRs_Voter.sldd processmodel.m

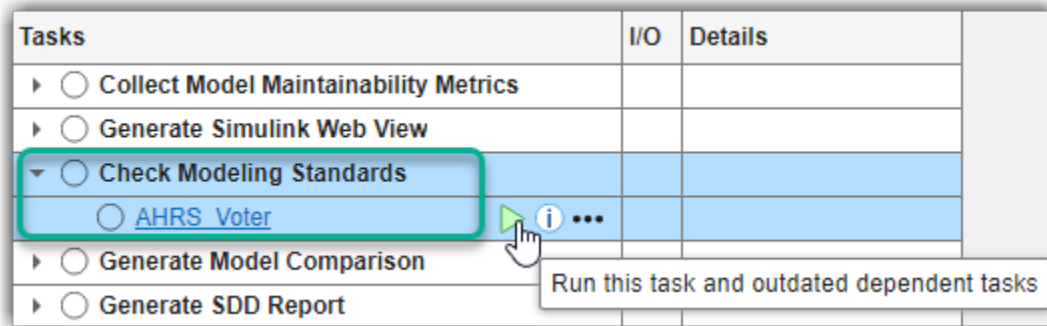
Task Iterations

The `IterationQuery` property of a task defines which artifacts a task iterates over and therefore how often the task runs. For example, by default, the **Check Modeling Standards** task uses the iteration query `padv.builtin.query.FindModels` to run one time for each model in the project. Although tasks iterate over artifacts, tasks do not automatically use those artifacts as inputs unless those artifacts are specified by the task input queries. Queries find artifacts in your project based on search criteria like the artifact type, project label, file path, and other characteristics. For more information, see “Find Artifacts with Queries” on page 2-23.

To change which artifacts a task iterates over, you can specify a different value for the `IterationQuery` property. For example, to have the **Check Modeling Standards** task only iterate over models that have `Voter` in their file path, you can modify the iteration query for the task.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());  
% Specify which set of artifacts to run for  
maTask.IterationQuery = ...  
    padv.builtin.query.FindModels(IncludePath = 'Voter');
```

The task iterations appear below the task title in the **Tasks** column in Process Advisor. If the iteration query does not return results, the task no longer appears in Process Advisor.



Tasks	I/O	Details
▶ ○ Collect Model Maintainability Metrics		
▶ ○ Generate Simulink Web View		
▼ ○ Check Modeling Standards		
○ AHRS Voter		
▶ ○ Generate Model Comparison		
▶ ○ Generate SDD Report		

See Also

[addInputQueries](#) | [addTask](#) | [padv.Task](#) | [Process Advisor](#) | [runprocess](#)

Related Examples

- “Add Tasks to Process” on page 2-13
- “Create Custom Tasks” on page 2-28
- “Define Task Relationships” on page 2-21
- “Overview of Process Model” on page 2-9
- “Modify Default Process Model to Fit Your Process” on page 2-2

Define Task Relationships


With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by adding tasks to your process model and reconfiguring the task behavior to meet the needs of your specific process.

Typically, you have dependencies between your tasks or you want your tasks to run in a specific order. In your process model, you can specify the relationship between tasks by using either the `dependsOn` method or the `runsAfter` method.

- **dependsOn** specifies that a task depends on inputs or results from another task to run successfully. When you run the downstream task, the build system automatically runs the upstream task first.
- **runsAfter** specifies a preferred task execution order for tasks that do not depend on each other. When both tasks are in the queue of tasks for the build system to run, the build system runs those tasks in your preferred task execution order when possible.

Open Process Model

You can specify the relationships between tasks by editing the process model file for the project. If you do not have a project or process model, see “Automate and Run Tasks with Process Advisor” on page 1-2 to get started.

- 1 Open the project that contains your files.
- 2 Open Process Advisor. On the **Project** tab, in the **Tools** section, click **Process Advisor**.
- 3 Edit the process model by clicking the **Edit** button  in the toolstrip.

Specify Relationships

Suppose you want to perform these tasks in the following order:

- 1 Check modeling standards with the built-in task `RunModelStandards`.
- 2 Generate code with the built-in task `GenerateCode`.
- 3 Inspect the generated code with the built-in task `RunCodeInspection`.

The following process model adds those tasks to the process model and specifies the relationships between those tasks by using the `runsAfter` and `dependsOn` methods.

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

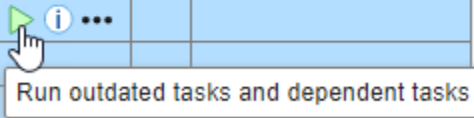
    % Add Tasks
    modelAdvisorTask = pm.addTask(padv.builtin.task.RunModelStandards);
    generateCodeTask = pm.addTask(padv.builtin.task.GenerateCode);
    inspectCodeTask = pm.addTask(padv.builtin.task.RunCodeInspection);

    % Define Task Relationships
    generateCodeTask.runsAfter(modelAdvisorTask);
    inspectCodeTask.dependsOn(generateCodeTask);
```

end

The code generation task should *run after* the Model Advisor task because code generation does not depend on the results or outputs from Model Advisor. The code inspection task *depends on* the generated code to have code to inspect and cannot run successfully without the outputs from code generation. In Process Advisor, if you point to the run button for a task that depends on another task, Process Advisor highlights that dependency. If you run a downstream task, like **Inspect Code**, the upstream dependencies, like **Generate Code**, run automatically.

Tasks	I/O	Details
<input type="checkbox"/> Check Modeling Standards		
<input type="checkbox"/> AHRS_Voter		
<input type="checkbox"/> Actuator_Control		
<input type="checkbox"/> Flight_Control		
<input type="checkbox"/> InnerLoop_Control		
<input type="checkbox"/> OuterLoop_Control		
<input type="checkbox"/> Generate Code		
<input type="checkbox"/> AHRS_Voter		
<input type="checkbox"/> Actuator_Control		
<input type="checkbox"/> Flight_Control		
<input type="checkbox"/> InnerLoop_Control		
<input type="checkbox"/> OuterLoop_Control		
<input type="checkbox"/> Inspect Code		
<input type="checkbox"/> AHRS_Voter		
<input type="checkbox"/> Actuator_Control		
<input type="checkbox"/> Flight_Control		
<input type="checkbox"/> InnerLoop_Control		
<input type="checkbox"/> OuterLoop_Control		



See Also

dependsOn | padv.ProcessModel | padv.Task | Process Advisor | runprocess | runsAfter

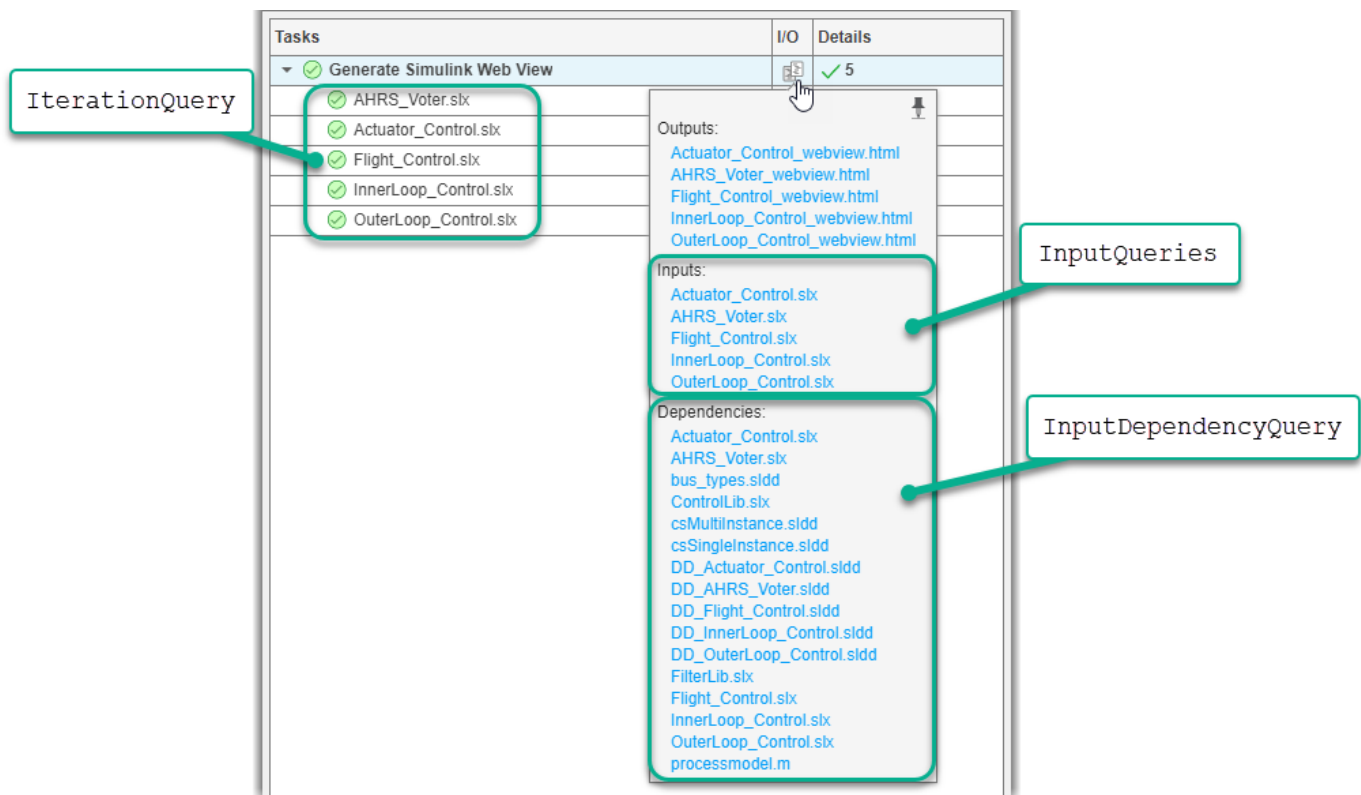
Related Examples

- “Add Tasks to Process” on page 2-13
- “Overview of Process Model” on page 2-9
- “Modify Default Process Model to Fit Your Process” on page 2-2

Find Artifacts with Queries

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by adding tasks to your process model and using queries to find the relevant artifacts for your tasks and process. There are built-in queries for finding common artifacts like models, requirements, and test cases, but you can also create your own custom queries. Typically, you use queries to find artifacts for your tasks to iterate over, use as task inputs, or use to find additional artifacts that your task inputs depend on.

In Process Advisor, the **Tasks** column shows the artifacts that the task iterates over. When you point to task results in the **I/O** column, you can see the task inputs and input dependencies. You define the task iterations, inputs, and input dependencies by specifying the associated task properties using queries.



Built-In Queries

The support package has built-in queries that can find specific sets of artifacts in your project. You can use the queries when you define your process, but note that you can only use certain queries as an input query (InputQueries) or iteration query (IterationQuery) for a task.

Query	Returns	Iteration Query	Input Query
padv.builtin.query.FindArtifacts	Artifacts that meet specified criteria	✓	Only when the query property InProject is false.

Query	Returns	Iteration Query	Input Query
<code>padv.builtin.query.FindCodeForModel</code>	Generated code files and <code>buildInfo.mat</code> for a model	✓	✓
<code>padv.builtin.query.FindDesignModels</code>	Units and components in project	✓	
<code>padv.builtin.query.FindExternalCodeCache</code>	External code cache files in project		✓
<code>padv.builtin.query.FindFileWithAddress</code>	File at the specified address	Only when the query property <code>TrackArtifacts</code> is true.	✓
<code>padv.builtin.query.FindFilesWithLabel</code>	Files with specific project label	✓	
<code>padv.builtin.query.FindMAJustificationFileForModel</code>	Find Model Advisor justification files	✓	✓
<code>padv.builtin.query.FindModels</code>	Models	✓	Only when the query property <code>InProject</code> is false.
<code>padv.builtin.query.FindModelsWithLabel</code>	Models with specific project label	✓	
<code>padv.builtin.query.FindModelsWithTestCases</code>	Models associated with a test case	✓	
<code>padv.builtin.query.FindProjectFile</code>	Project file	✓	✓
<code>padv.builtin.query.FindRefModels</code>	Referenced models	✓	
<code>padv.builtin.query.FindRequirements</code>	Requirement sets	✓	Only when the query property <code>InProject</code> is false.
<code>padv.builtin.query.FindRequirementsForModel</code>	Requirements associated with model	✓	✓
<code>padv.builtin.query.FindTestCasesForModel</code>	Test cases associated with model	✓	✓
<code>padv.builtin.query.FindTopModels</code>	Top models	✓	✓
<code>padv.builtin.query.FindUnits</code>	Units in the project	✓	✓

Query	Returns	Iteration Query	Input Query
<code>padv.builtin.query.GetDependentArtifacts</code>	Dependent artifacts for artifact		✓
<code>padv.builtin.query.GetIterationArtifact</code>	Artifact that the task is iterating over		✓
<code>padv.builtin.query.GetOutputsOfDependentTask</code>	Outputs from immediate predecessor task		✓

Custom Queries

If you need to find artifacts that are not already covered by built-in queries, you can use custom queries in your process model. Depending on what you want your custom query to do, there are different approaches. For more information, see “Create Custom Queries” on page 2-39.

Valid Artifact Types

The support package creates a digital thread to monitor artifacts in your project and analyze their relationships. The digital thread is a set of metadata that allows Process Advisor and its build system to detect changes to the project and identify outdated task results. However, the digital thread only tracks changes to specific types of artifacts shown below. If you use custom queries that return unsupported artifact types, the digital thread cannot detect changes to those artifacts, which can limit the ability of Process Advisor to identify outdated tasks results. To see a list of the files the digital thread is tracking in your project, see “Find Artifacts that Digital Thread Tracks”.

The digital thread tracks the following types of artifacts. If an artifact in your project is represented by a `padv.Artifact` object with any other artifact type, such as `"other_file"`, changes to that artifact do not cause tasks to become outdated.

Category	Artifact Type	Description
MATLAB	<code>"m_class"</code>	MATLAB class
	<code>"m_file"</code>	MATLAB file
	<code>"m_func"</code>	MATLAB function
	<code>"m_method"</code>	MATLAB class method
	<code>"m_property"</code>	MATLAB class property
Model Advisor	<code>"ma_config_file"</code>	Model Advisor configuration file
	<code>"ma_justification_file"</code>	Model Advisor justification file
Process Advisor	<code>"padv_dep_artifacts"</code>	Related artifacts that current artifact depends on
	<code>"padv_output_file"</code>	Process Advisor output file
Project	<code>"project"</code>	Current project file
Requirements	<code>"mwreq_item"</code>	Requirement (since R2024b)
	<code>"sl_req"</code>	Requirement (for R2024a and earlier)

Category	Artifact Type	Description
Stateflow®	"sl_req_file"	Requirement file
	"sl_req_table"	Requirements Table
	"sf_chart"	Stateflow chart
	"sf_graphical_fcn"	Stateflow graphical function
	"sf_group"	Stateflow group
	"sf_state"	Stateflow state
Simulink	"sf_state_transition_chart"	Stateflow state transition chart
	"sf_truth_table"	Stateflow truth table
	"sl_block_diagram"	Block diagram
	"sl_data_dictionary_file"	Data dictionary file
	"sl_embedded_matlab_fcn"	MATLAB function
	"sl_block_diagram"	Block diagram
	"sl_library_file"	Library file
	"sl_model_file"	Simulink model file
	"sl_protected_model_file"	Protected Simulink model file
	"sl_subsystem"	Subsystem
System Composer™	"sl_subsystem_file"	Subsystem file
	"sl_subsystem"	Subsystem
	"zc_block_diagram"	System Composer architecture
Tests	"zc_component"	System Composer architecture component
	"zc_file"	System Composer architecture file
	"harness_info_file"	Harness info file
Tests	"sl_harness_block_diagram"	Harness block diagram
	"sl_harness_file"	Test harness file
	"sl_test_case"	Simulink Test case
	"sl_test_case_result"	Simulink Test case result
	"sl_test_file"	Simulink Test file
	"sl_test_iteration"	Simulink Test iteration
	"sl_test_iteration_result"	Simulink Test iteration result
	"sl_test_report_file"	Simulink Test result report
	"sl_test_result_file"	Simulink Test result file

Category	Artifact Type	Description
	"sl_test_resultset"	Simulink Test result set
	"sl_test_seq"	Test Sequence
	"sl_test_suite"	Simulink Test suite
	"sl_test_suite_result"	Simulink Test suite result

Dynamically Resolve Paths with Tokens

To dynamically resolve paths for artifacts, directories, and other process information, you can use tokens as placeholders. The default process model and built-in task source code use the following tokens.

Token	Description
\$INPUTARTIFACT\$	Input artifact for task
\$ITERATIONARTIFACT\$	Current artifact that the task is acting on
\$PWD\$	Current working directory
\$TIMESTAMP\$	Current date and time in the format 'yyyy_mm_dd_HH_MM_ss'
\$PROJECTROOT\$	Root folder of project
\$TASKNAME\$	Task name or title
\$DEFAULTOUTPUTDIR\$	Default output directory for the process model
\$ROOTITERATIONARTIFACT\$	Root-level artifact for the iteration artifact

You can use these tokens in your process model, but note that:

- The output directory of a task cannot be \$PROJECTROOT\$.
- The pipeline generator, `padv.pipeline.generatePipeline`, does not support the tokens \$PWD\$, \$TIMESTAMP\$, and \$INPUTARTIFACT\$.

Inside the `run` method and `dryRun` method for a task, you can resolve tokens to their absolute path by using the `resolvePath` method. For example:

```
reportPath = convertStringsToChars(obj.resolvePath(obj.ReportPath));
```

See Also

Related Examples

- “Create Custom Queries” on page 2-39
- “Overview of Process Model” on page 2-9

Create Custom Tasks

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by adding built-in and custom tasks to your process. The support package contains several built-in tasks that you can reconfigure and use to perform steps in your process, but if you need to perform other actions or always want to use a reconfigured version of a built-in task, you can create and add custom tasks to your process model.

Depending on what you want your custom task to do, there are different approaches:

- For basic MATLAB script execution — Use the `addTask` function to create a new task and use the `Action` argument to specify a function handle for a function that runs the script. See “Custom Task that Runs Existing Script” on page 2-28.
- For more complex tasks — Create a MATLAB class that inherits from either one of the “Built-In Tasks” on page 2-14 or the superclass `padv.Task` and then override class properties and methods to fit your needs. See “Custom Task for Specialized Functionality” on page 2-29 and “Example Custom Tasks” on page 2-33. To view the source code for a built-in task, use the `open` function.

To add custom tasks to your process, you need to edit the process model file for your project. If you do not have a project or process model, see “Automate and Run Tasks with Process Advisor” on page 1-2 to get started.

Custom Task that Runs Existing Script

If your custom task only needs to run an existing MATLAB script, you can edit your process model to specify which script to run by using the `Action` argument for the `addTask` function.

For example, suppose that you have a script, `myScript.m`, that you want a custom task to run. You can use the `addTask` function to add a new task to your process model. The `Action` argument specifies the function that the task runs. In your `processmodel.m` file, you can specify:

```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

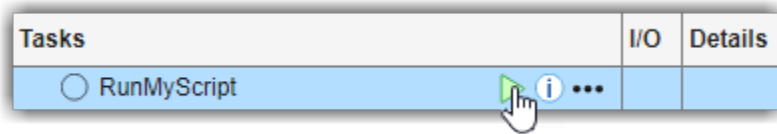
    addTask(pm, "RunMyScript", Action = @runMyScript);

end

function taskResult = runMyScript(~)
    run("myScript.m");
    taskResult = padv.TaskResult;
end
```

"RunMyScript" is the name for the new task. `@runMyScript` is the function handle for the function that you define inside the `processmodel.m` file. `padv.TaskResult` defines the results for the task.

You can run the script as a task in Process Advisor.



Custom Task for Specialized Functionality

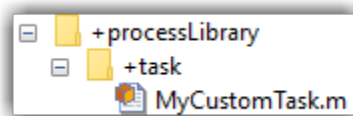
To create a task that performs a custom functionality, you need to:

- 1 Create a new MATLAB class.
- 2 Inherit from either a built-in task or the superclass `padv.Task`.
- 3 Specify the task name and, optionally, other task properties.
- 4 Keep or override the `run` method that defines the action that the task performs.

Create New MATLAB Class

Create a new MATLAB class in your project.

Note that namespaces can help you organize the class definition files for your custom tasks. For example, in the root of your project you can create a folder `+processLibrary` with a subfolder `+task` and save your class in that folder.



To share your custom tasks across multiple process models in different projects, consider creating a referenced project that contains your folders and class definition files. Your main projects can then use the referenced project as a shared process library.

Choose Superclass for Custom Task

Your MATLAB class can inherit from either:

- One of the “Built-In Tasks” on page 2-14 — Use this approach when there is a built-in task that is similar to the custom task that you want to create. When you inherit from a built-in task, like `padv.builtin.task.RunModelStandards`, your custom task inherits the functionality of that task, but then you can override the properties and methods of the class to fit your needs. For information on the built-in tasks, see “Built-In Tasks” on page 2-14.
- The superclass `padv.Task` — Use this approach if your custom task needs to perform a step that is not similar to a built-in task. `padv.Task` is the base class of the built-in tasks, so you must completely define the inputs, functionality, and outputs of the task.

If you are inheriting from a built-in task, you can replace the contents of your class file with the following example code. The code inherits from the built-in task `padv.builtin.task.RunModelStandards`, but you can replace those lines of code to inherit from a different built-in task instead.

```
classdef MyCustomTask < padv.builtin.task.RunModelStandards
    % task definition goes here
```

```

methods
function obj = MyCustomTask(options)
arguments
options.Name = "MyCustomTask";
options.Title = "My Custom Task";
end
obj@padv.builtin.task.RunModelStandards(Name = options.Name);
obj.Title = options.Title;
end
end
end

```

If you are inheriting from `padv.Task`, you can replace the contents of your class file with the following example code. The code finds the models in the project by using the iteration query `padv.builtin.query.FindModels` and specifies those models as task inputs by using the input query `padv.builtin.query.GetIterationArtifact`. The code calls the constructor of the superclass `padv.Task`. For information on superclass constructors, see “Design Subclass Constructors”.

```

classdef MyCustomTask < padv.Task
methods
function obj = MyCustomTask(options)
arguments
% unique identifier for task
options.Name = "MyCustomTask";
% artifacts the task iterates over
options.IterationQuery = "padv.builtin.query.FindModels";
% input artifacts for the task
options.InputQueries = "padv.builtin.query.GetIterationArtifact";
% where the task outputs artifacts
options.OutputDirectory = fullfile(...
'$DEFAULTOUTPUTDIR$', 'my_custom_task_results');
end

% Calling constructor of superclass padv.Task
obj@padv.Task(options.Name,...
IterationQuery=options.IterationQuery,...
InputQueries=options.InputQueries);
obj.OutputDirectory = options.OutputDirectory;
end

function taskResult = run(obj,input)
% "input" is a cell array of input artifacts
% length(input) = number of input queries

% class definition goes here

% specify results from task using padv.TaskResult
taskResult = padv.TaskResult;
taskResult.Status = padv.TaskStatus.Pass;
% taskResult.Status = padv.TaskStatus.Fail;
% taskResult.Status = padv.TaskStatus.Error;
end
end
end

```

Specify Task Properties

Specify the `Name` property for your task and, optionally, other task properties.

You must specify a `Name` because the `Name` is the unique identifier for the task. Specifying other class properties is optional, but can help you define the task behavior. The following table lists common class properties that you often specify for a custom task. For information on other class properties, see “Built-In Tasks” on page 2-14 or `padv.Task`. For information on how tasks and queries define your process, see “Overview of Process Model” on page 2-9.

Property	Description
<code>Name</code> (required)	Unique identifier for task
<code>IterationQuery</code>	Artifacts the task iterates over By default, custom tasks run one time for the project.
<code>InputQueries</code>	Inputs to the task
<code>InputDependencyQuery</code>	Artifacts that the task inputs depend on Typically, you specify <code>InputDependencyQuery</code> as <code>padv.builtin.query.GetDependentArtifacts</code> to get the dependent artifacts for each task input.
<code>OutputDirectory</code>	Directory where the task outputs artifacts If you do not specify <code>OutputDirectory</code> for a custom task, the build system stores task outputs in the <code>DefaultOutputDirectory</code> specified by <code>padv.ProcessModel</code> .

Keep or Override run Method

The `run` method defines the action that your custom task performs. For examples of how to override the `run` method, see “Example Custom Tasks” on page 2-33.

Make sure to use the same method signature as the class that you inherit from. In the method signature, the `input` argument is a cell array that contains the input artifacts from your input queries. Each element in `input` corresponds to each input query that you specify.

For example, if you only specify one input query, `padv.builtin.query.GetIterationArtifact`, and you are iterating over each model in the project, you can use the first element of `input`, `input{1}`, to perform an action on each model in the project:

```
function taskResult = run(obj,input)
    % Before the task loads models,
    % save a list of the models that are already loaded.
    loadedModels = get_param(Simulink.allBlockDiagrams(), 'Name');

    % identify model name
    % "input" is a cell array of input artifacts
    % First input query gets iteration artifact (a model)
    model = input{1}; % get padv.Artifact from first input query
    modelName = padv.util.getModelName(model);

    % Example task that loads model and displays information
```

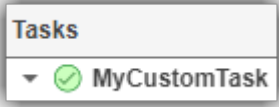
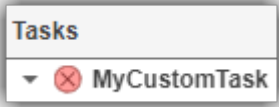
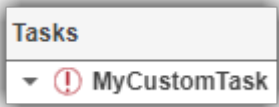
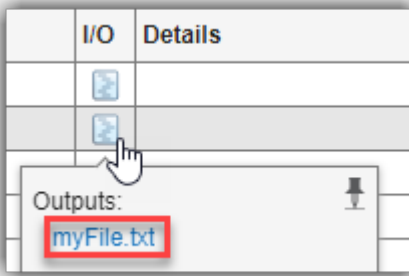
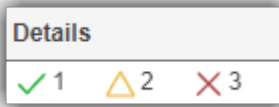
```

load_system(modelName);
disp(modelName);
disp('Data Dictionaries:');
disp(Simulink.data.dictionary.getOpenDictionaryPaths)

% specify results from task using padv.TaskResult
taskResult = padv.TaskResult;
taskResult.Status = padv.TaskStatus.Pass;
% taskResult.Status = padv.TaskStatus.Fail;
% taskResult.Status = padv.TaskStatus.Error;

%% Close models that were loaded by this task.
padv.util.closeModelsLoadedByTask(...
    PreviouslyLoadedModels=loadedModels)
end
    
```

The run method must return a `padv.TaskResult` object. Process Advisor uses the `padv.TaskResult` object to assess the status of your custom task. The task result properties `Status`, `OutputPaths`, and `ResultValues` correspond to the **Tasks**, **I/O**, and **Details** columns in Process Advisor:

Example Code	Appearance in Process Advisor
<pre>taskResult.Status = padv.TaskStatus.Pass</pre>	
<pre>taskResult.Status = padv.TaskStatus.Fail</pre>	
<pre>taskResult.Status = padv.TaskStatus.Error</pre>	
<pre>taskResult.OutputPaths=string(... fullfile("PA_Results","myFile.txt"));</pre>	
<pre>taskResult.ResultValues.Pass = 1; taskResult.ResultValues.Warn = 2; taskResult.ResultValues.Fail = 3;</pre>	

Additionally, you can also override the `dryRun` method to specify how your custom task evaluates task inputs and generates representative outputs for quick process model tests. For more information, see “Dry Run Tasks to Test Process Model” on page 2-66.

Add Custom Task to Process

Add your custom task to your process model by using the `addTask` function. For example, to add a custom task named `MyCustomTask` that is saved in a `+task` subfolder inside a `+processLibrary` folder:

```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    addTask(pm, processLibrary.task.MyCustomTask);
end
```

The custom task appears in the **Tasks** column in Process Advisor.

Tasks	I/O	Details
▼ <input type="radio"/> MyCustomTask		
<input type="radio"/> AHRs_Voter.slx		
<input type="radio"/> Actuator_Control.slx		
<input type="radio"/> Flight_Control.slx		
<input type="radio"/> InnerLoop_Control.slx		
<input type="radio"/> OuterLoop_Control.slx		

Example Custom Tasks

Perform Post-Processing on Task Results

You can use custom tasks to perform pre-processing or post-processing actions. For example, suppose you want to run Model Advisor and if checks generate a failure or a warning, you want the task to fail. There are no built-in tasks that perform this exact functionality by default, but the built-in task `padv.builtin.task.RunModelStandards` runs Model Advisor and the task fails if a check generates a failure.

You can use a custom task to create your own version of `padv.builtin.task.RunModelStandards` that overrides the results from the task to specify that if a Model Advisor check returns a warning, the task should also fail.

This example shows a custom task that inherits from the built-in task `padv.builtin.task.RunModelStandards`, overrides the input queries to use the file `sampleChecks.json` as the Model Advisor configuration file, and extends the `run` method of the built-in task to fail the task if Model Advisor returns warnings.

```
classdef MyRunModelStandards < padv.builtin.task.RunModelStandards
    % RunModelStandards, but use my Model Advisor configuration file
    % and fail the task when there are warnings from Model Advisor checks

    methods
        function obj = MyRunModelStandards(options)

            arguments
                options.Name = "MyRunModelStandards";
                options.Title = "My Check Modeling Standards";
            end

            obj@padv.builtin.task.RunModelStandards(Name = options.Name);
            obj.Title = options.Title;
            % specify current model (iteration artifact) and
            % Model Advisor configuration file as inputs to the task
            obj.addInputQueries([padv.builtin.query.GetIterationArtifact,...
                padv.builtin.query.FindFileWithAddress(...
                    Type = 'ma_config_file',...
                    Path = fullfile('tools','sampleChecks.json'))]);

        end

        function taskResult = run(obj,input)

            % use RunModelStandards to run Model Advisor
            taskResult = run@padv.builtin.task.RunModelStandards(obj,input);
            % If checks for a model fail, then the status will be
            % set to fail.

            % But you can extend the built-in task to specify that
            % if checks for a model generate a warning, then the
            % task status will also be set to fail.
            if taskResult.ResultValues.Warn > 0
                taskResult.Status=padv.TaskStatus.Fail;
            end

        end

    end

end
```

Note In this example, the run method of the custom task extends the run method of the built-in task by calling it from within the custom task run method. But you can also reimplement the run method for a custom task to implement your own version of the run method. For more information and common class designs, see “Modify Inherited Methods”.

Run Custom Task for Project

Suppose that you want to return a list of the data dictionaries in your project. There are no built-in tasks that perform this functionality, so you can create a custom task that inherits directly from the base class `padv.Task` and use the arguments to specify the behavior of the custom task.


```

classdef ListAllDataDictionaries < padv.Task

    methods
        function obj = ListAllDataDictionaries(options)

            arguments
                options.InputQueries = padv.builtin.query.FindArtifacts(...
                    ArtifactType="sl_data_dictionary_file");
                options.Name = "ListAllDataDictionaries";
            end
            inputQueries = options.InputQueries;
            obj@padv.Task(options.Name, ...
                Title = "My Custom Task for SLDD files", ...
                InputQueries = inputQueries, ...
                DescriptionText = "My Custom Task for SLDD files", ...
                Licenses={});

        end

        function taskResult = run(~, input)
            % Print names of SLDDs
            disp([input{1}.Alias]')
            taskResult = padv.TaskResult;
            taskResult.Status = padv.TaskStatus.Pass;
            taskResult.ResultValues.Pass = 1;
        end
    end
end
end

```

In the custom task, you can find the data dictionaries in the project by using the query `padv.builtin.query.FindArtifacts` and specifying the query as one of the `InputQueries` for the task. In the run function, you can specify the action that the task performs and specify the task results, in a format that Process Advisor can recognize, by using a `padv.TaskResult` object. The `input` is a cell array of input artifacts that the build system automatically creates based on the `InputQueries` that you specify. In this example, the first cell in `input` is an array of `padv.Artifact` objects that represent the data dictionaries in the project. The `disp` function can display the aliases of the data dictionaries in the MATLAB Command Window. When you specify the task result `Status`, that sets the task status in the **Tasks** column in Process Advisor. `ResultValues.Pass` sets the number of passing results in the **Details** column in Process Advisor.

Tasks	I/O	Details
✓ My Custom Task for SLDD files	📄	✓ 1

```

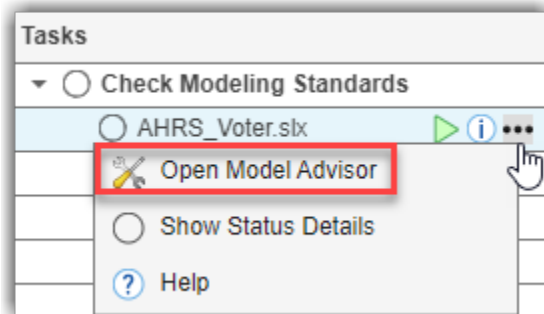
#### Tasks to run:
#### ListAllDataDictionaries::ProcessAdvisorExample.prj
#####
## Starting task ListAllDataDictionaries::ProcessAdvisorExample.prj
"DD_AHRS_Voter"
"DD_Actuator_Control"
"DD_Flight_Control"
"DD_InnerLoop_Control"
"DD_OuterLoop_Control"
"csMultiInstance"
"csSingleInstance"
"bus_types"

```

Specify Tool for Custom Task

When you point to a task in the Process Advisor app, you can click the ellipsis (...) to view more options. For built-in tasks, you have the option to launch a tool or multiple tools associated with the

task. For example, the built-in task **Check Modeling Standards** allows you to directly open Model Advisor for the model that the task iteration runs on.



You can associate a tool with the options menu for a task by specifying the property `LaunchToolAction` as a function handle that launches that tool. For example, suppose you have a custom task that runs on each model in the project and you want the task to launch Dependency Analyzer for the model. For `LaunchToolAction`, specify the handle to a function that launches Dependency Analyzer. The function that launches the tool has two inputs, `obj` and `artifact`, and must return a `result` structure with the status of the tool launch action, `ToolLaunched`.

```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    customTask = addTask(pm,"MyCustomTask",...
        IterationQuery = padv.builtin.query.FindModels,...
        InputQueries = padv.builtin.query.GetIterationArtifact,...
        LaunchToolAction=@openDependencyAnalyzer);

end

function result = openDependencyAnalyzer(obj, artifact)
    result = struct('ToolLaunched', false);
    % handle non-model task iterations / abstract tasks
    if isempty(artifact)
        result.message = 'Open the tool for an artifact listed under the task title.';
        return;
    end
    % identify model name
    modelName = padv.util.getModelName(artifact);
    % open Dependency Analyzer for model
    depview(modelName)
    result.ToolLaunched = true;
end
```

Specify Inputs That Can Make Task Outdated

Suppose that you want to create a custom task that analyzes specific Excel® files in your project and you want the task to become outdated when you make changes to those files. You can find the files by using the built-in query `padv.builtin.query.FindArtifacts`. In this example, the task uses the `IncludePathRegex` argument of the query to find Excel files (`.xlsx`) with file names that begin

with HLR_. The task uses that query to define the task iterations (IterationQuery) and task inputs (InputQueries). The task iterates over these files and checks for the presence of specific sheets named StepUp and StepDown. If the Excel file has those sheets, the task passes. Otherwise, the task fails. The task automatically becomes outdated if you make a change to any of the Excel files that the query finds.

```

classdef CheckExcelSheetNames < padv.Task
    methods
        function obj = CheckExcelSheetNames(options)
            arguments
                % unique identifier for task
                options.Name = "CheckExcelSheetNames";
                % artifacts the task iterates over
                % in this case, Excel files that begin with "HLR_"
                options.IterationQuery = padv.builtin.query.FindArtifacts(...
                    IncludePathRegex = "HLR_.*\.xlsx");
                % input artifacts for the task
                % in this case, the same as the iteration artifacts
                options.InputQueries = "padv.builtin.query.GetIterationArtifact";
                % where the task outputs artifacts
                options.OutputDirectory = fullfile(...
                    '$DEFAULTOUTPUTDIR$', 'excel_status_results');
            end
            % Calling constructor of superclass padv.Task
            obj@padv.Task(options.Name,...
                IterationQuery=options.IterationQuery,...
                InputQueries=options.InputQueries);
            obj.OutputDirectory = options.OutputDirectory;
        end
        function taskResult = run(obj,input)
            % specify results from task using padv.TaskResult
            taskResult = padv.TaskResult;
            % Get the sheet names for the sheets in the current spreadsheet
            % "input" is a cell array of task input artifacts
            a = input{1}.ArtifactAddress;
            fa = a.getFileAddress;
            sheets = sheetnames(fa);
            % Check if sheets for both "StepUp" and "StepDown" are present in
            % the spreadsheet
            if ismember("StepUp", sheets) && ismember("StepDown", sheets)
                disp('Both the "StepUp" and "StepDown" sheets are present.');
```

Ignore Changes to Specific Task Outputs

You can turn off change tracking for a specific artifact by specifying the Track property of the artifact address as false. The artifact address is stored in the ArtifactAddress property of a padv.Artifact object.

For example, the following custom task inherits from the built-in task `DetectDesignErrors`, but overrides the `run` method to turn off change tracking for the output report. The custom task identifies the report by iterating over each task output, checking if the artifact has the same report format as the task, and then specifying the `Track` property for the artifact address.

```
classdef MyDetectDesignErrors < padv.builtin.task.DetectDesignErrors
    % Detect design errors, but ignore changes to generated report files
    methods

        function obj = MyDetectDesignErrors(options)
            arguments
                options.Name = "MyDetectDesignErrors";
                options.Title = "My Detect Design Errors";
            end
            obj@padv.builtin.task.DetectDesignErrors(Name = options.Name);
            obj.Title = options.Title;
        end

        function taskResult = run(obj,input)

            % use DetectDesignErrors to run Design Verifier
            taskResult = run@padv.builtin.task.DetectDesignErrors(obj,input);

            % for each task output, check if it's a report
            for i = 1:length(taskResult.OutputArtifacts)
                artifact = taskResult.OutputArtifacts(i);
                artifactAddress = artifact.ArtifactAddress;
                fileAddress = artifactAddress.getFileAddress;
                if contains(fileAddress, obj.ReportFormat, IgnoreCase=true)
                    % if the task output is a report, turn off change tracking for the report
                    artifactAddress.Track = false;
                end
            end
        end
    end
end
```

For more information, see “Exclude Files from Change Tracking in Process Advisor” on page 2-59.

See Also

`addTask` | `padv.ProcessModel` | `padv.Task` | Process Advisor | `runprocess`

Related Examples

- “Add Tasks to Process” on page 2-13
- “Find Artifacts with Queries” on page 2-23
- “Overview of Process Model” on page 2-9
- “Modify Default Process Model to Fit Your Process” on page 2-2

Create Custom Queries

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by using a process model. You can use queries to find artifacts relevant to your tasks and processes. The support package contains several built-in queries that you can reconfigure and use to find artifacts in your project, but if you need to perform other actions or always want to use a reconfigured version of a built-in query, you can create and add custom queries to your process model.

To find artifacts in your project, you can use the built-in queries that ship with the support package or you can create your own custom queries. Use the built-in queries where possible. If your use case requires custom queries, use the following steps to create a custom query. Note that to reconfigure the functionality of a built-in task, your custom queries can inherit from a built-in query.

After you create a custom query, you can use that query as an input query for a task to modify or filter the task inputs.

Choose Superclass for Custom Query

There are two ways to define custom queries:

- Inherit from a built-in query — Use this approach when there is a built-in query that is similar to the custom query that you want to create. When you inherit from a built-in query, like `padv.builtin.query.FindArtifacts`, your custom query inherits the functionality of that query, but then you can override the properties and methods of the class to fit your needs.
- Inherit from `padv.Query` — Use this approach if your custom query needs to find artifacts in a way that is not similar to a built-in query. `padv.Query` is the base class of the built-in queries, so you must completely define the functionality of the query.

Define and Use Custom Query in Process

- 1 Create a new MATLAB class in your project.

Tip Namespaces can help you organize the class definition files for your custom queries. In the root of your project, create a folder `+processLibrary` with a subfolder `+query` and save your class in that folder.

To share your custom queries across multiple process models in different projects, consider creating a referenced project that contains your folders and class definition files. Your main projects can then use the referenced project as a shared process library.

- 2 Use one of these approaches to define your custom query:

- If you are inheriting from a built-in query, you can replace the contents of your class file with this example code:

```
classdef MyCustomQuery<padv.builtin.query.FindArtifacts
    % query definition goes in this class
    % by default, this query finds all artifacts in the project
    methods
        function obj = MyCustomQuery(NameValueArgs)
            arguments
                NameValueArgs.Name = "MyCustomQuery";
```

```

end
end
end
end

```

This example query inherits from the built-in query `padv.builtin.query.FindArtifacts`, but you can change that line of code to inherit from another built-in query. Use the properties of the query to specify which sets of artifacts you want the query to return. If you want to override the run method for a built-in query, check which input arguments the run method for the built-in query accepts and use the same method signature inside your custom query. For more information, see “Built-In Queries” on page 2-23.

- If you are inheriting from `padv.Query`, you can replace the contents of your class file with this example code:

```

classdef MyCustomQuery < padv.Query

    methods
        function obj = MyCustomQuery(NameValueArgs)
            obj@padv.Query("MyCustomQuery");
        end

        function artifacts = run(obj,~)
            artifacts = padv.Artifact.empty;
            % Core functionality of the query goes here
            % artifacts = padv.Artifact(artifactType,...
            % padv.util.ArtifactAddress(fullfile(fileparts)));
        end
    end
end

```

A query must have:

- a unique name, specified using the `Name` property
- a `run` function that returns either a `padv.Artifact` object or an array of `padv.Artifact` objects. For more information, see `padv.Artifact` and “Example Custom Queries” on page 2-41.

Note The digital thread only tracks changes to specific types of artifacts. For information on supported artifact types, see “Valid Artifact Types” on page 2-25. If there is an artifact in your project that the `padv.builtin.query.FindArtifacts` query does not find, the digital thread cannot detect changes to that artifact. If you create custom queries that return `padv.Artifact` objects with unsupported artifact types, the digital thread will not detect changes to those artifacts. This behavior can impact whether changes to these artifacts cause a task to be marked as outdated. To see a list of the files the digital thread is tracking in your project, see “Find Artifacts that Digital Thread Tracks”.

- 3 You can test your custom query in the MATLAB Command Window executing the `run` function. Note that your project needs to be open for the query to find artifacts. For example, for a query `MyCustomQuery` saved in the namespace `processLibrary.query`:

```
run(processLibrary.query.MyCustomQuery)
```

- 4 You can use your custom query in your process model. For example, you can control which artifacts a task iterates over by using your custom query as the iteration query for a task:

```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    t = addTask(pm, "MyCustomTask", ...
        IterationQuery = processLibrary.query.MyCustomQuery);

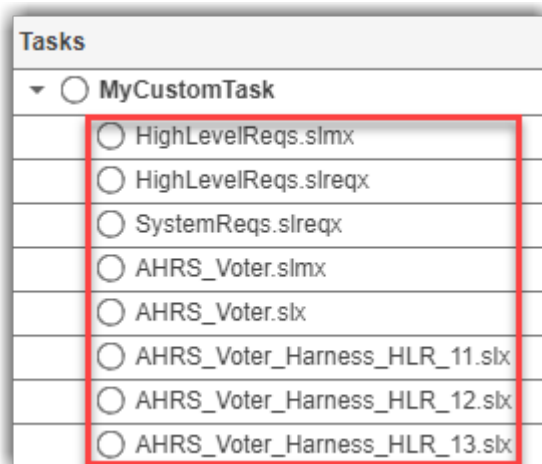
end
```

This example assumes that you saved your class file in the +query folder inside the +processLibrary folder.

- 5 You can confirm which artifacts your task iterates over by opening Process Advisor. In the MATLAB Command Window, enter:

```
processAdvisorWindow
```

The artifacts that the task iterates over appear under the task name in the **Tasks** column.



Example Custom Queries

Run Task on Data Dictionaries in Project

Suppose you want to find each of the data dictionaries in your project. There are no built-in queries that perform this functionality by default, but there is a built-in query `padv.builtin.query.FindArtifacts` that can find artifacts that meet certain search criteria. Effectively you can create your own version of the built-in query, but specialized to only find data dictionaries. You can create a class-based, custom query that inherits from `padv.builtin.query.FindArtifacts` and specifies the `ArtifactType` argument as a Simulink data dictionary.

```
classdef FindSLDDs < padv.builtin.query.FindArtifacts
    %FindSLDDs This query is like FindArtifacts,
    % but only returns data dictionaries.
    methods
        function obj = FindSLDDs(NameValueArgs)
```

```

        arguments
            NameValueArgs.ArtifactType string = "sl_data_dictionary_file";
            NameValueArgs.Name = "FindSLDDs";
        end
        obj.ArtifactType = NameValueArgs.ArtifactType;
    end
end
end

```

The example class FindSLDDs inherits its properties and run function from the built-in query `padv.builtin.query.FindArtifacts`, but specifies a unique Name and ArtifactType. The ArtifactType is specified as `sl_data_dictionary_file` because that is the artifact type associated with Simulink data dictionary files. For a list of valid artifact types, see `padv.builtin.query.FindArtifacts`.

You can have a task run once for each data dictionary in your project by using the custom query as the iteration query for the task.

```

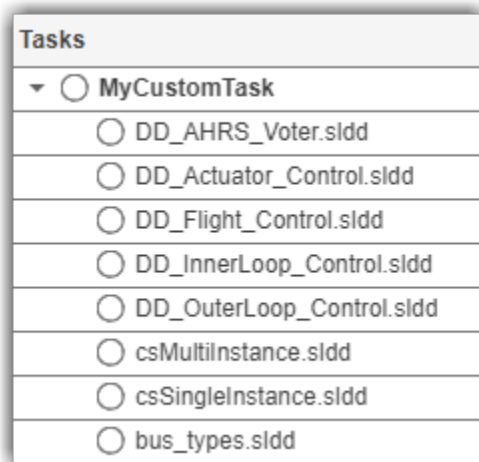
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    t = addTask(pm, "MyCustomTask", ...
        IterationQuery = processLibrary.query.FindSLDDs);

end

```



Sort Artifacts in Specific Order

By default, queries sort artifacts alphabetically by the artifact address. If you want your query to sort artifacts in a different order, you can override the internal `sortArtifacts` method in a subclass that defines a custom sort behavior. For example:

```

classdef FindFileSorted < padv.builtin.query.FindArtifacts
    methods
        function obj = FindFileSorted(options)

```



```

arguments
    options.ArtifactType string
    options.IncludeLabel string
    options.ExcludeLabel string
    options.IncludePath string
    options.ExcludePath string
    options.InProject boolean
    options.FilterSubFileArtifacts boolean
end
fwdoptions = namedargs2cell(options);
obj@padv.builtin.query.FindArtifacts(fwdoptions{:});
end
end
methods(Access = protected)
% Overload the default sort artifacts logic, in this case
% Sorting artifacts based upon their string length rather than
% Alphabetically
function sortedArtifacts = sortArtifacts(~, artifacts)
    if isempty(artifacts)
        sortedArtifacts = artifacts;
        return;
    end
    namesToSort = arrayfun(@(art) art.ArtifactAddress.getFileAddress, artifacts);
    [~, idx] = sort(strlength(namesToSort));
    sortedArtifacts = artifacts(idx);
end
end
end
end

```

Note If you override `sortArtifacts`, make sure that your implementation only changes the order of the artifacts, not the data type or structure. Do not use `sortArtifacts` to add or remove artifacts from the query results.

Run Validation Scripts on Spreadsheets

Suppose that your project contains several Excel spreadsheets and that for each spreadsheet, you have a validation script with the same name as the spreadsheet. You can find the validation scripts by using a custom query and then you can run the validation script on each spreadsheet by using a custom task. For example, the following example custom query searches through the artifacts in the project to find if there are any scripts that have the same name as the iteration artifact.

```

classdef FindValidationFiles < padv.Query

    methods
        function obj = FindValidationFiles(NameValueArgs)
            arguments
                NameValueArgs.Name string = string.empty;
                NameValueArgs.Title string = "Find validation files";
            end

            obj@padv.Query(NameValueArgs.Name, Title=NameValueArgs.Title);

            % Named Query
            obj.CanBeUsedAsInputQuery = true;
            obj.CanBeUsedAsIterationQuery = true;
        end
    end
end

```

```
end

function paArtifact = run(~,iterationArtifact)
    paArtifact = padv.Artifact.empty;

    % Get Name of iteration artifact
    [~,name] = fileparts(iterationArtifact.ArtifactAddress.getFileAddress());
    % Find validation script with same name as iteration artifact
    validationFileName = strcat(name, ".m");
    filePath = which(validationFileName);
    if ~isempty(filePath)
        paArtifact = padv.Artifact("xls_validation_file",filePath);
    end
end

end
end
```

In the process model, you can add the custom query as an input query for your custom task so that if you make a change to the validation script, the task iteration for that spreadsheet automatically becomes outdated. For example, this example process model uses the built-in query `FindArtifacts` to find the spreadsheets, specifies that a custom task named `RunValidationScript` must iterate over each spreadsheet returned by the `FindArtifacts` query, and then adds the custom query as an input query for the task.

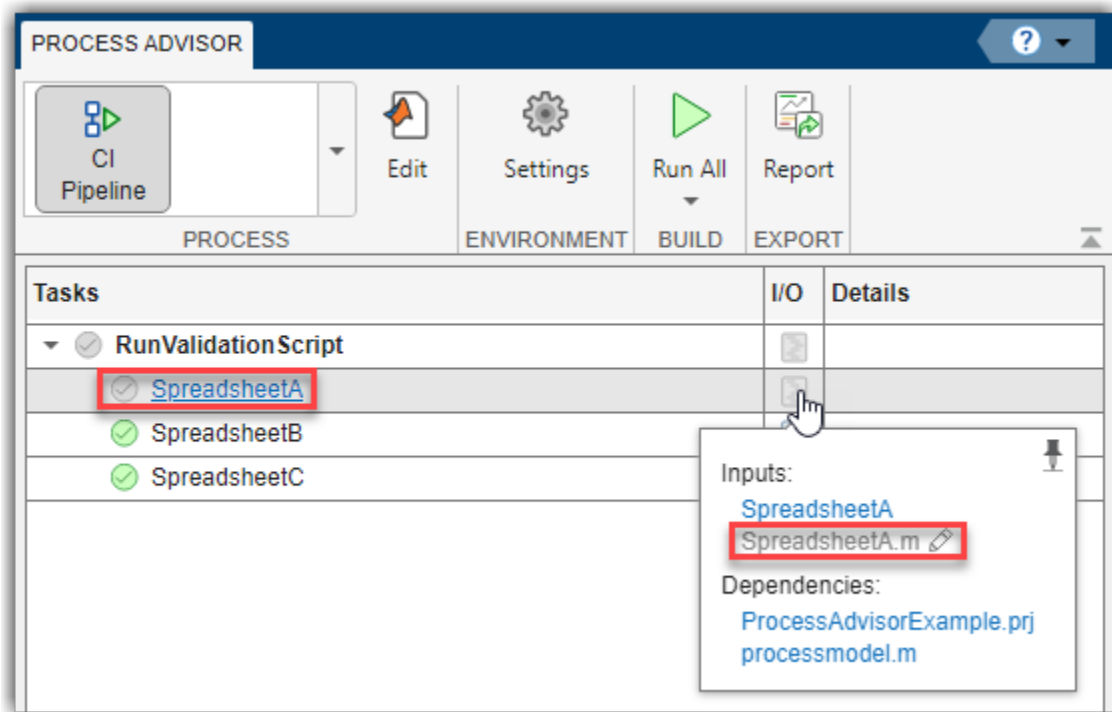
```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    findSpreadsheets = padv.builtin.query.FindArtifacts(IncludePathRegex = "Spreadsheet.*\.xlsx");
    validationTask = pm.addTask(RunValidationScript(IterationQuery=findSpreadsheets));
    validationTask.addInputQueries(FindValidationFiles);

end
```

The validation task automatically becomes outdated if you make changes to the validation scripts because you specified the custom query `FindValidationFiles` as an input query for the task.



See Also

[padv.Artifact](#) | [padv.ProcessModel](#) | [padv.Query](#) | [Process Advisor](#) | [runprocess](#)

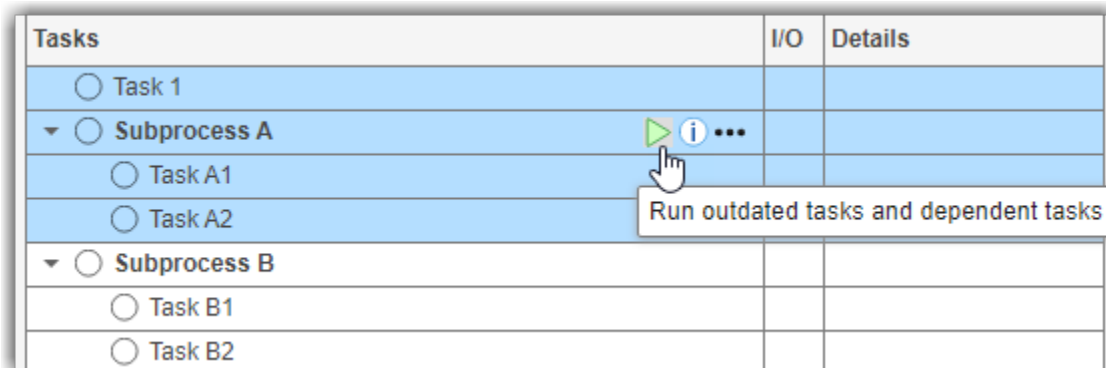
Related Examples

- “Find Artifacts with Queries” on page 2-23
- “Overview of Process Model” on page 2-9
- “Modify Default Process Model to Fit Your Process” on page 2-2
- “Reconfigure Task Behavior” on page 2-17

Group Tasks with Subprocesses


With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by using a process model. Within a process, you can use subprocesses to group related tasks, create a hierarchy of tasks, and share parts of your overall process. A *subprocess* is a self-contained sequence of tasks, inside a process or other subprocess, that can run standalone.

Tasks	I/O	Details
<input type="radio"/> Task 1		
▼ <input type="radio"/> Subprocess A		
<input type="radio"/> Task A1		
<input type="radio"/> Task A2		
▼ <input type="radio"/> Subprocess B		
<input type="radio"/> Task B1		
<input type="radio"/> Task B2		



Open Process Model

You can group tasks into subprocesses by editing the process model file for your project. If you do not have a project or process model, see “Automate and Run Tasks with Process Advisor” on page 1-2 to get started.

- 1 Open the project that contains your files.
- 2 Open Process Advisor. On the **Project** tab, in the **Tools** section, click **Process Advisor**.
- 3 Edit the process model by clicking the **Edit** button  in the toolstrip.

Add Tasks to Specific Subprocess

To group the tasks in your process:

- 1 In the process model, you can add a subprocess by using `addSubprocess` on your process model object.


```
spA = pm.addSubprocess("Subprocess A");
```
- 2 Instead of adding your tasks directly to your process model object, add your tasks to a specific subprocess by using `addTask`.


```
tA1 = spA.addTask("Task A1");
tA2 = spA.addTask("Task A2");
```
- 3 You can use the `dependsOn` and `runsAfter` methods to define the relationships between tasks and subprocesses in your process.

For example, the following process model defines a process in which Task 1 runs, then Subprocess A, and then Subprocess B.

```
function processmodel(pm)
    % Defines the project's processmodel
```

```

arguments
    pm padv.ProcessModel
end

t1 = pm.addTask("Task 1");

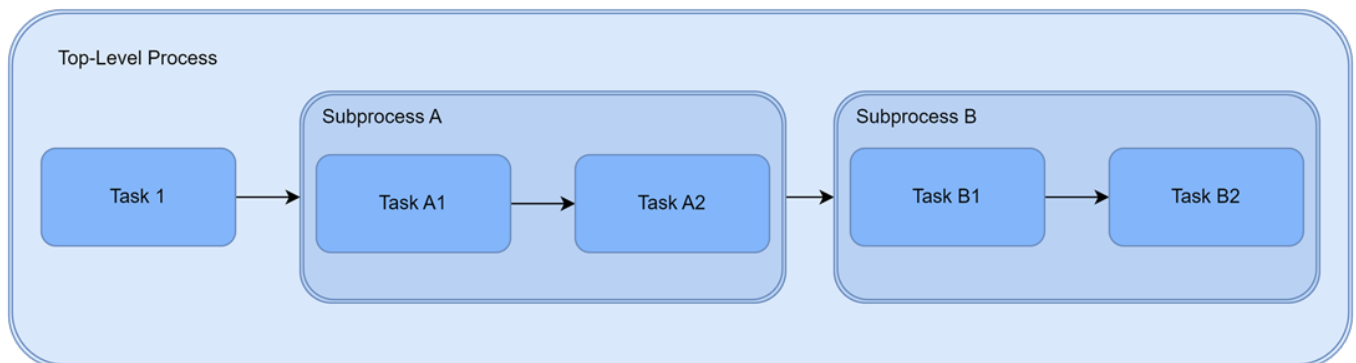
spA = pm.addSubprocess("Subprocess A");
    tA1 = spA.addTask("Task A1");
    tA2 = spA.addTask("Task A2");
spB = pm.addSubprocess("Subprocess B");
    tB1 = spB.addTask("Task B1");
    tB2 = spB.addTask("Task B2");

% Relationships
spA.dependsOn(t1);
    tA2.dependsOn(tA1);
spB.dependsOn(spA);
    tB2.dependsOn(tB1);

end

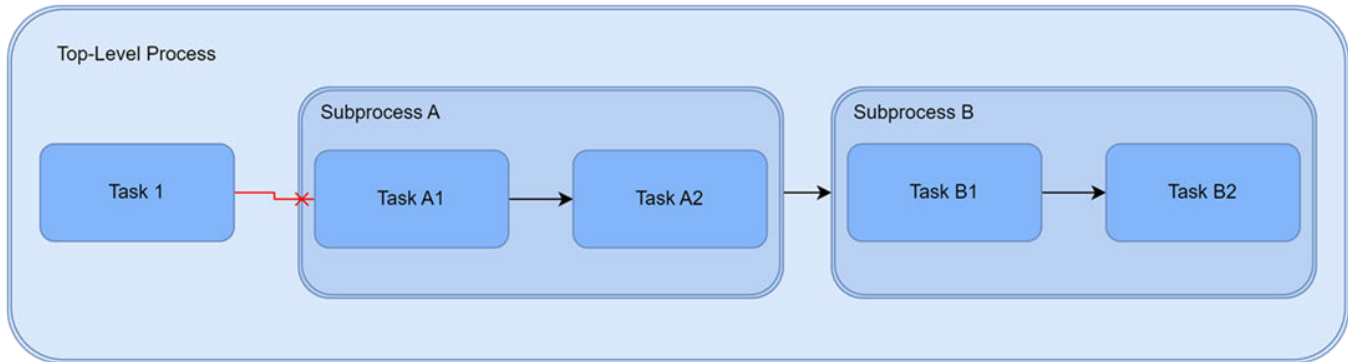
```

The build system executes each of the tasks inside a subprocess before exiting the subprocess. The following diagram shows a graphical representation of the relationships defined by that process model.



Considerations for Subprocess Boundaries

The relationships that you specify in the process model cannot cross any subprocess boundaries. For example, in the previous process model, you cannot directly specify that Task A1 depends on Task 1 because that relationship would enter into Subprocess A, crossing the subprocess boundary.



In this case, you need to create a relationship between the Task 1 and Subprocess A instead.

Example Process Model with Subprocesses

To access an example process model that groups tasks into subprocesses for **Model Verification** and **Code Verification**, enter `processAdvisorExampleStart(Subprocess = true)` at the command line.

Tasks	I/O	Details
<ul style="list-style-type: none"> ▼ ○ Model Verification <ul style="list-style-type: none"> ▶ ○ Collect Model Maintainability Metrics ▶ ○ Generate Simulink Web View ▶ ○ Check Modeling Standards ▶ ○ Generate SDD Report ▶ ○ Run Tests ▶ ○ Merge Test Results ▶ ○ Collect Model Testing Metrics ▼ ○ Code Verification <ul style="list-style-type: none"> ▶ ○ Generate Code ▶ ○ Check Coding Standards ▶ ○ Prove Code Quality 		

Run outdated tasks and dependent tasks

See Also

`padv.Subprocess`

Related Examples

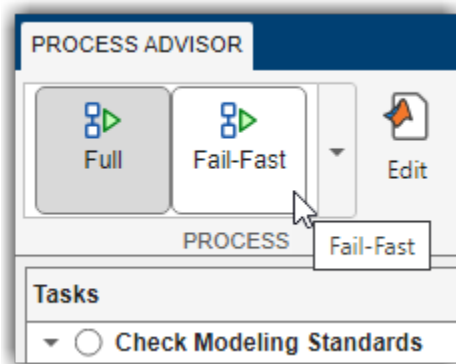
- “Overview of Process Model” on page 2-9
- “Manage Multiple Build and Verification Workflows Using Processes” on page 2-49

Manage Multiple Build and Verification Workflows Using Processes

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by using a process model. Inside your process model, you can define multiple processes for the different build and verification workflows, environments, and other situations that your team needs a defined process for. A process is a group of tasks or subprocesses inside your process model. For example, you can create separate processes for:


- Smoke testing with fail-fast tasks
- Local prequalification
- CI builds
- Different stages of the development process
- Different product readiness levels

Processes allow you to have multiple build and verification processes standardized and available to your team, with the tasks configured appropriately for that specific workflow. In Process Advisor, you can select which process you want to use from the **Processes** gallery in the toolstrip. APIs like the `runprocess` function also allow you to specify which Process to run.



Open Process Model

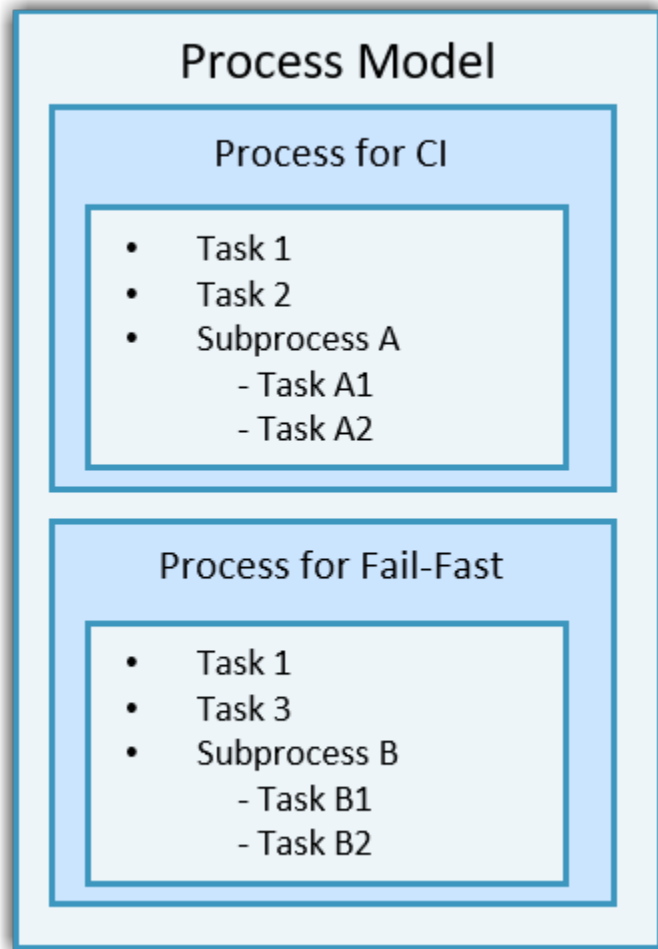
You can create multiple processes by editing the process model file for your project. If you do not have a project or process model, see “Automate and Run Tasks with Process Advisor” on page 1-2 to get started.

- 1 Open the project that contains your files.
- 2 Open Process Advisor. On the **Project** tab, in the **Tools** section, click **Process Advisor**.
- 3 Edit the process model by clicking the **Edit** button  in the toolstrip.

Overview of Processes

Your process model can contain multiple processes and each process can contain tasks and subprocesses. For example, you can have one process for your full qualification process and another process for fail-fast, local prequalification. Each of those processes can contain tasks and or subprocesses of tasks. Additionally, you can share tasks and subprocesses across multiple processes.

You can use the `padv.Process` methods like `addTask`, `addSubprocess`, `addDependsOnRelationship`, and `addRunsAfterRelationship` inside the process model to define the tasks, subprocesses, and relationships for your process.



Define New Processes

Add Processes

To add a new process inside your process model, use the method `addProcess`.

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    % Add processes
    processA = pm.addProcess("A");
    processB = pm.addProcess("B");

end
```


Define Processes Using Process-Specific Methods

When you define multiple processes inside your process model, use the `padv.Process` methods to add tasks, subprocesses, and relationships directly to your process. Unlike the methods for `padv.ProcessModel`, which add tasks and subprocesses to the default process inside your process model, the `padv.Process` methods allow you to specify which specific process you want to add a task, subprocess, or relationship to. For example, if you have multiple processes and want to specify a dependency between two tasks inside a process, use the `padv.Process` method `addDependsOnRelationship` to specify that dependency. The method `addDependsOnRelationship` accepts the process name as an input argument. Using process-specific methods is especially important if you share tasks across multiple processes and need to define different relationships to that task within each process.

The class `padv.Process` has several methods that you can use to customize the process. For more information, see `padv.Process`.

<code>addTask</code>	Add task to process <code>myProcess.addTask("myTask");</code>
<code>addSubprocess</code>	Add subprocess to process <code>myProcess.addSubprocess("mySubprocess");</code>
<code>addDependsOnRelationship</code>	Create dependency between two tasks <code>myProcess.addDependsOnRelationship(... Source=taskB, ... Dependency=taskA);</code> The build system always runs the Dependency task before the Source task.
<code>addRunsAfterRelationship</code>	Specify predecessor for task <code>myProcess.addRunsAfterRelationship(... Source=taskB, ... Predecessor=taskA);</code> When you run your process, the build system runs the Predecessor task before the Source task when possible.

Add Tasks and Organize Tasks Within Process

You can add tasks directly to a specific process by using the `addTask` method for `padv.Process`. You can also use subprocesses to organize tasks within your process. For example, this process model uses the `padv.Process` methods to add example tasks and subprocesses to specific processes in the process model.

```
function processmodel(pm)
    % This function defines a process model for a project by setting up processes,
    % subprocesses, and tasks within those processes.

    arguments
        pm padv.ProcessModel
    end
```

```
% --- Processes ---
% Add processes
processA = pm.addProcess("A");
processB = pm.addProcess("B");

% --- Tasks ---
% Create example tasks
task1 = padv.Task("task1");
task2 = padv.Task("task2");
task3 = padv.Task("task3");
taskA1 = padv.Task("taskA1");
taskA2 = padv.Task("taskA2");
taskB1 = padv.Task("taskB1");
taskB2 = padv.Task("taskB2");

% --- Subprocesses ---
% Add subprocesses to parent process
subprocessA = processA.addSubprocess("subprocessA"); % Add to process A
subprocessB = processB.addSubprocess("subprocessB"); % Add to process B

% --- Add Tasks to Processes ---
processA.addTask(task1); % Add task1 to process A
processA.addTask(task2); % Add task2 to process A
processB.addTask(task1); % Reuse task1 in process B
processB.addTask(task3); % Add task3 to process B

% --- Add Tasks to Subprocesses ---
subprocessA.addTask(taskA1); % Add taskA1 to subprocessA under process A
subprocessA.addTask(taskA2); % Add taskA2 to subprocessA under process A
subprocessB.addTask(taskB1); % Add taskB1 to subprocessB under process B
subprocessB.addTask(taskB2); % Add taskB2 to subprocessB under process B

end
```

Note By default, if you do not add processes to a process model, the process model automatically creates a default process, "CIPipeline", for you. If you add tasks and subprocesses directly to the `padv.ProcessModel` object, you are actually adding those tasks and subprocesses to an intermediate, default process. By default, the default process is "CIPipeline".

You can access the `padv.Process` object that represents the default process by using the method `findProcess` inside the process model.

```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    pm.addTask("MyCustomTask");
    processCI = pm.findProcess("CIPipeline");

end
```

Add Task Relationships to Process

To specify a preferred task execution order inside a specific process, use the `padv.Process` method `addRunsAfterRelationship`. For example, for each process, you can specify that a shared task, `task1`, should run after a specific task in that process.

```
% Create Dependencies Within Specific Process
processA.addRunsAfterRelationship(Source = task2,...
    Predecessor = task1);
processB.addRunsAfterRelationship(Source = task3,...
    Predecessor = task1);
```

You can add a dependency between tasks inside a specific process by using the `padv.Process` method `addDependsOnRelationship`. For example, you can specify that for `processA`, `task2` depends on `task1` and cannot run without `task1` running first.

```
% Add dependency between tasks inside Process A
processA.addDependsOnRelationship(...
    Source = task2,...
    Dependency = task1);
```

Example Process Model with Multiple Processes

Suppose that you have two Model Advisor configuration files:

- `allChecks.json` — Contains all of the Model Advisor checks that you want to run
- `quickChecks.json` — Contains a subset of your Model Advisor checks for fast-fail checking

For your full process, you can add an instance of the `RunModelStandards` task that runs using `allChecks.json`. For your "fail-fast" process, you can add an instance of the `RunModelStandards` task that runs using `quickChecks.json`. Note that this code shares the query object, `findModels`, across the tasks to improve performance. By sharing the query object, the build system can avoid re-running the `padv.builtin.query.FindModels` query. For more information, see "Best Practices for Process Model Authoring" on page 2-56.

```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    fullProcess = pm.addProcess("Full");
    failfastProcess = pm.addProcess("Fail-Fast");

    % Define Shared Query and Add Shared Query to Process Model
    findModels = padv.builtin.query.FindModels(Name="ModelsQuery");
    pm.addQuery(findModels);

    % Add Full Model Advisor Checks Task to CI Process
    % (uses allChecks.json MA config file)
    taskFullMA = fullProcess.addTask(...
        padv.builtin.task.RunModelStandards(...
            Name = "fullMATask",...
            IterationQuery=findModels));
    taskFullMA.addInputQueries(...
        padv.builtin.query.FindFileWithAddress(...
```

```

    Type='ma_config_file', Path=fullfile('tools','allChecks.json')));

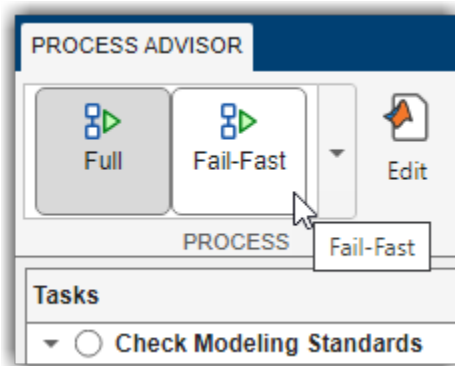
% Add Quick Checks Task to Fail-Fast Process
% (uses quickChecks.json MA config file)
taskFailFastMA = failfastProcess.addTask(...
    padv.builtin.task.RunModelStandards(...
        Name = "quickChecksTask",...
        IterationQuery=findModels));
taskFailFastMA.Title = "Check Modeling Standards (subset)";
taskFailFastMA.addInputQueries(...
    padv.builtin.query.FindFileWithAddress(...
        Type='ma_config_file', Path=fullfile('tools','quickChecks.json')));

end

```

Use Specific Process

In Process Advisor, you can select which process you want to use from the **Processes** gallery in the toolstrip. By default, processes appear in the order that you define them in the process model.



APIs like the `runprocess` function and `processadvisor` function also allow you to specify which process to use.

```

runprocess(Process = "Fail-Fast")
processadvisor("AHRV_Voter", "Fail-Fast")

```

Default Process Behavior

By default, the default process is the first process that you add to your process model. When you open Process Advisor for the first time, the **Tasks** column shows the tasks in your default process. APIs like the `runprocess` function use the default process unless you specify which process to use. You can specify a different default process by overriding the `DefaultProcessId` property of the `padv.ProcessModel` object inside your process model.

```

pm.DefaultProcessId = "Fail-Fast";

```

To identify which process is the default process for the process model, you can use the `getprocess` function in the MATLAB Command Window.

```

getprocess().DefaultProcessId

```

The first time that you open Process Advisor, the app opens to the default process. Otherwise, Process Advisor re-opens your last opened process. To force the next Process Advisor session to open the

default process instead of the last opened process, you can reset your user settings from the MATLAB Command Window.

```
us = padv.UserSettings.get();  
us.resetToDefaultValues();
```

See Also

`padv.Process` | `padv.Subprocess`

Related Examples

- “Group Tasks with Subprocesses” on page 2-46
- “Overview of Process Model” on page 2-9

Best Practices for Process Model Authoring

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by using a process model. When you define your process model, consider the following process modeling best practices that you can use to maintain your process model, handle dependencies, and improve process model loading times.


Keep Process Model File in Project Root

By default, the build system automatically creates a process model file in the root folder of the project. If possible, keep your process model file in the root folder of the project so that the build system can detect changes to the file and mark tasks as outdated.


Make Sure Only One Process Model File on Path

To avoid unexpected behavior, make sure only one `processmodel` file is on the path. You can instruct the build system to detect when there are multiple process model files on the path. For more information, see the property `DetectMultipleProcessModels` for `padv.ProjectSettings`.

Review Untracked Dependencies

If you make a change to an untracked input or output file, Process Advisor does not mark the task as outdated. Make sure that task inputs or outputs that appear as **Untracked**  do not need to be tracked to maintain the task status and result information that you need for your project.

By default, the build system generates a warning for untracked I/O files. To change build system behavior when there are untracked I/O files, you can specify the project setting **Untracked dependency behavior** as either:

- "Allow" — Do not generate warnings or errors for untracked I/O files.
- "Warn" — Generate a warning if a task has untracked I/O files. In Process Advisor, the **I/O** column shows a warning icon .
- "Error" — Generate an error if a task has untracked I/O files.

For more information, see “Specify Settings for Process Advisor and Build System” on page 1-16. For more information on untracked files and change tracking, see “Exclude Files from Change Tracking in Process Advisor” on page 2-59.

Share Queries Across Tasks

You can improve Process Advisor load times by sharing query instances across your process model. If multiple tasks in the process model use the same iteration query, you can update your code to share a single query object instance across these tasks. For example, if multiple tasks use `FindModels` as an iteration query, you can create a `FindModels` object and use that object as the iteration query for those tasks:

Functionality
<pre>taskA = pm.addTask("taskA",... IterationQuery = padv.builtin.query.FindModels); taskB = pm.addTask("taskB",... IterationQuery = padv.builtin.query.FindModels);</pre>
Use This Instead
<pre>sharedModelsQuery = padv.builtin.query.FindModels(... Name="SharedModelsQuery"); taskA = pm.addTask("taskA",... IterationQuery = sharedModelsQuery); taskB = pm.addTask("taskB",... IterationQuery = sharedModelsQuery);</pre>

Parent Queries

A query can use the results of another query by specifying that query as a parent. For example, the built-in query `padv.builtin.query.FindModelsWithTestCases` uses the built-in query `padv.builtin.query.FindModels` as a parent query to initially find the models in the project and then the built-in query `padv.builtin.query.FindModelsWithTestCases` itself finds the test cases associated with those models.

You can specify a parent query for the following built-in queries by using the `Parent` name-value argument:

- `padv.builtin.query.FindCodeForModel`
- `padv.builtin.query.FindMAJustificationFileForModel`
- `padv.builtin.query.FindModelsWithTestCases`
- `padv.builtin.query.FindRequirementsForModel`
- `padv.builtin.query.FindTestCasesForModel`

For example, multiple built-in queries use the built-in query `padv.builtin.query.FindModels` as a parent query. For iteration queries, the build system runs the parent query first to find the initial set of artifacts that the child query can run on. You can use the `Query` argument to specify a shared parent query.

```
findModels = padv.builtin.query.FindModels(Name="ModelsQuery");
findModelsWithTests = padv.builtin.query.FindModelsWithTestCases(...
  Name = "ModelsWithTests",...
  Parent = findModels);
findTestsForModel = padv.builtin.query.FindTestCasesForModel(...
  Name = "TestsForModel",...
  Parent = findModels);
pm.addQuery(findModels);
pm.addQuery(findModelsWithTests);
pm.addQuery(findTestsForModel);
```

See Also

Related Examples

- “Best Practices for Effective Builds” on page 3-32
- “Overview of Process Model” on page 2-9
- “Specify Settings for Process Advisor and Build System” on page 1-16

Exclude Files from Change Tracking in Process Advisor

When you use CI/CD Automation for Simulink Check, the support package creates a digital thread for your project to capture and analyze certain project artifacts and their relationships. Process Advisor and its build system monitor the digital thread to detect changes and identify outdated task results. By default, any change to an artifact or its dependencies makes related task results outdated.

However, you can exclude specific artifacts from change tracking if you know that certain artifacts change regularly and do not impact task validity. This can help you reduce unnecessary task re-runs while maintaining the task status and result information that you need for your project. You can modify the change tracking behavior for the:

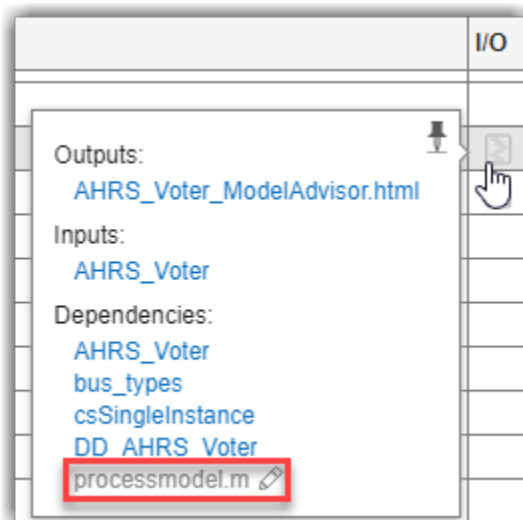
- “Process Model” on page 2-59
- “Task Inputs” on page 2-60
- “Task Outputs” on page 2-60

Make sure to review the files that you exclude from change tracking. For more information, see “Handling Untracked Dependencies” on page 2-62.

Process Model

If you do not want changes to the process model to make task results outdated, you can open the **Settings** in Process Advisor and clear **Add process model as dependency** in the user settings. For more information, see “Specify Settings for Process Advisor and Build System” on page 1-16 and `padv.UserSettings`.

By default, if you make a change to the process model file, Process Advisor marks task results as outdated because the tasks in the updated process model might not match the tasks that generated the task results from the previous version of the process model. This behavior occurs because Process Advisor automatically adds the process model as a dependency for each task to help make sure that tasks in the updated process model match the tasks that generated the previous task results.



Task Inputs

Turn Tracking Off for Query Artifacts

If you find your task inputs by using the built-in query `padv.builtin.query.FindFileWithAddress`, you can turn off change tracking for the artifacts that the query returns by specifying the query property `TrackArtifacts` as `false`. The file that the query returns is untracked and if you make a change to the file, Process Advisor does not mark the task as outdated.

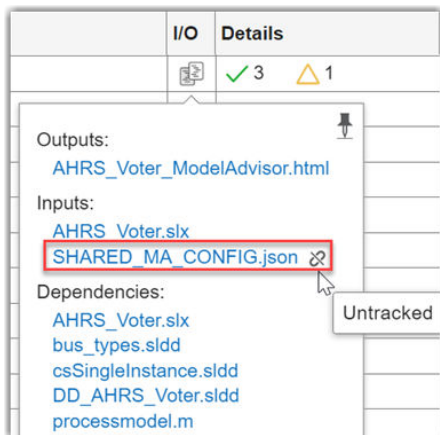
```
padv.builtin.query.FindFileWithAddress(...
Type='ma_config_file', Path=which('sampleChecks.json'),...
TrackArtifacts = false)
```

Tracking Behavior for Artifacts Outside Project

You can use artifacts outside your project as inputs to your tasks, but changes to those files are untracked by default. For example, if you have a shared Model Advisor configuration file, `SHARED_MA_CONFIG.json`, that is outside your project, you can add the file as an input to the **Check Modeling Standards** task.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
maTask.addInputQueries(padv.builtin.query.FindFileWithAddress( ...
    Type='ma_config_file', Path=which('SHARED_MA_CONFIG.json')));
```

In the Process Advisor **I/O** column, the file appears as **Untracked** because you cannot track changes to files outside the project. If you make a change to an untracked file, Process Advisor does not mark the task as outdated.



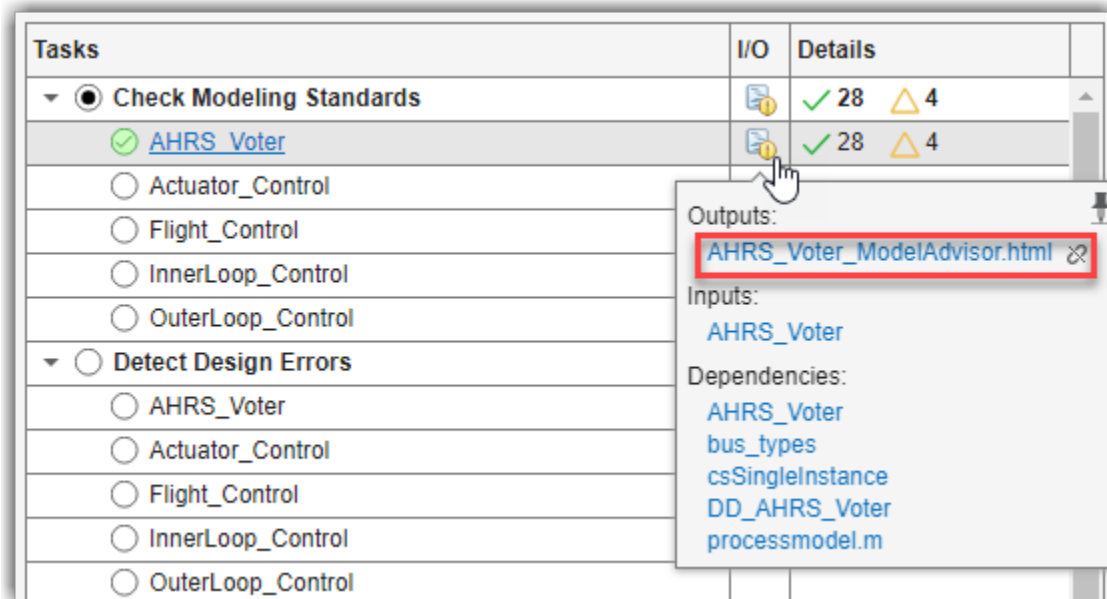
Task Outputs

Turn Tracking Off for All Task Outputs

If you do not want Process Advisor to mark a task as outdated when you make changes to task outputs, you can turn off change tracking for those task outputs. In your process model, specify the task property `TrackOutputs` as `false`.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
maTask.TrackOutputs = false;
```

In the Process Advisor **I/O** column, the outputs appear as **Untracked**. If you make a change to an untracked file, the Process Advisor does not mark the task as outdated.



Turn Tracking Off for Specific Task Outputs

You can turn off change tracking for a specific artifact by specifying the `Track` property of the artifact address as `false`. The artifact address is stored in the `ArtifactAddress` property of a `padv.Artifact` object. The built-in queries typically return artifacts as `padv.Artifact` objects, but you can also manually define an artifact address for an artifact by using the utility function `padv.util.ArtifactAddress`.

You can ignore changes to specific task outputs by specifying the `Track` property inside a custom task. For example, the following custom task inherits from the built-in task `DetectDesignErrors`, but overrides the `run` method to turn off change tracking for the output report. The custom task identifies the report by iterating over each task output, checking if the artifact has the same report format as the task, and then specifying the `Track` property for the artifact address.


```
classdef MyDetectDesignErrors < padv.builtin.task.DetectDesignErrors
    % Detect design errors, but ignore changes to generated report files
    methods
        function obj = MyDetectDesignErrors(options)
            arguments
                options.Name = "MyDetectDesignErrors";
                options.Title = "My Detect Design Errors";
            end
            obj@padv.builtin.task.DetectDesignErrors(Name = options.Name);
            obj.Title = options.Title;
        end


        function taskResult = run(obj,input)

            % use DetectDesignErrors to run Design Verifier
            taskResult = run@padv.builtin.task.DetectDesignErrors(obj,input);
        end
    end
end
```

```
% for each task output, check if it's a report
for i = 1:length(taskResult.OutputArtifacts)
    artifact = taskResult.OutputArtifacts(i);
    artifactAddress = artifact.ArtifactAddress;
    fileAddress = artifactAddress.getFileAddress;
    if contains(fileAddress, obj.ReportFormat, IgnoreCase=true)
        % if the task output is a report, turn off change tracking for the report
        artifactAddress.Track = false;
    end
end
end
end
end
```

Handling Untracked Dependencies

By default, the Process Advisor generates a warning when you have untracked I/O files that impact your tasks. In the **I/O** column, Process Advisor shows a warning icon  for tasks that have untracked inputs or outputs. You can change this behavior by opening the **Settings** in Process Advisor and specifying the project setting **Untracked dependency behavior** to either allow, generate a warning, or generate an error if a task has untracked I/O files. For more information, see “Specify Settings for Process Advisor and Build System” on page 1-16 and `padv.ProjectSettings`.

If you make a change to an untracked input or output file, Process Advisor does not mark the task as outdated. Make sure that task inputs or outputs that appear as **Untracked**  do not need to be tracked to maintain the task status and result information that you need for your project.

See Also

`padv.Artifact` | `padv.builtin.query.FindFileWithAddress` | `padv.Task` |
`padv.ProjectSettings` | `padv.UserSettings` | `padv.util.ArtifactAddress`

Related Examples

- “Best Practices for Process Model Authoring” on page 2-56
- “Create Custom Tasks” on page 2-28
- “Specify Settings for Process Advisor and Build System” on page 1-16

Test Tasks and Queries

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process by adding tasks to a process model and using queries to find relevant artifacts like models, requirements, and test cases. If you are trying to debug or test a task or query, it can be helpful to run the task or query directly from the MATLAB Command Window. To test a task, you can find the ID for a specific task iteration and use the `runprocess` function to run that task iteration. To test a query, you can create an instance of the query and use the `run` method to get the artifacts that the query returned.

This example shows how to test a built-in query and then use the artifacts that the query returns to test a built-in task. For more information on built-in tasks and queries, see the “Built-In Tasks” on page 2-14 and “Built-In Queries” on page 2-23. To evaluate the task inputs and outputs defined by your process model, you can dry run tasks as shown in “Dry Run Tasks to Test Process Model” on page 2-66.

Open Project

Open a project. For this example, you can open the Process Advisor example project.

```
processAdvisorExampleStart
```

Find Artifacts Using Query

Suppose that you want to test the built-in query `padv.builtin.query.FindModels`.

- 1 In the MATLAB Command Window, create an instance of the query.

```
q = padv.builtin.query.FindModels;
```

- 2 To see which artifacts the query returns, run the query by using the `run` method.

```
artifacts = run(q)
```

```
artifacts =
```

```
1×5 Artifact array with properties:
```

```
Type
Parent
ArtifactAddress
Alias
```

In this example, the query returns the five models in the example project. If you open the `Alias` property, you can see the names of each of the models returned by the `padv.builtin.query.FindModels` query.

```
artifacts.Alias
```

- 3 To filter the artifacts returned by the query, you can modify the behavior of the query using the name-value arguments. For example, to exclude artifacts that contain `Control` in the file path, you would specify:

```
q = padv.builtin.query.FindModels(ExcludePath = "Control");
```

- 4 Re-run the query to see the updated query results.

```
artifacts = run(q)
```

```
artifacts =
```

```
Artifact with properties:
```

```
    Type: "sl_model_file"  
    Parent: [0x0 padv.Artifact]  
    ArtifactAddress: [1x1 padv.util.ArtifactAddress]  
    Alias: "AHRV_Voter.slx"
```

For this example, the query returns a single Simulink model, `AHRV_Voter.slx`, since `AHRV_Voter.slx` is the only model that does not contain `Control` in its file path.

```
artifacts.ArtifactAddress
```

```
ans =
```

```
ArtifactAddress
```

```
    FileAddress: "02_Models/AHRV_Voter/specification/AHRV_Voter.slx"  
    OwningProject: "ProcessAdvisorExample"  
    IsSubFileArtifact: 0
```

If the artifact is in a referenced project, the `OwningProject` returns the name of the referenced project. If you need to know which project contains an artifact, you can use the `getOwningProject` function on the artifact address object. For more information, see `padv.util.ArtifactAddress`.

Run Task for Specific Artifacts

Suppose that you want to run the task `padv.builtin.task.GenerateSimulinkWebView` on the `AHRV_Voter` model returned by a query.

You can run a specific task iteration by specifying the `Tasks` and `FilterArtifact` name-value arguments for the `runprocess` function.

```
runprocess(...  
Tasks = "padv.builtin.task.GenerateSimulinkWebView",...  
FilterArtifact = artifacts(1))
```

You can use the other name-value arguments of `runprocess` to specify how the task iteration runs. For example, `Force = true` forces the task iteration to run, even if the results are already up-to-date and `Isolation = true` has the task iteration run without running any of its dependencies.

```
runprocess(...  
Tasks = "padv.builtin.task.GenerateSimulinkWebView",...  
FilterArtifact = artifacts(1),...  
Force = true,...  
Isolation = true)
```

For more information, see `runprocess`.

See Also

`padv.Task` | `padv.Query` | `runprocess`

Related Examples

- “Overview of Process Model” on page 2-9
- “Add Tasks to Process” on page 2-13
- “Find Artifacts with Queries” on page 2-23
- “Dry Run Tasks to Test Process Model” on page 2-66

Dry Run Tasks to Test Process Model


With the CI/CD Automation for Simulink Check support package, you can define a development and verification process by adding tasks to a process model and using queries to find relevant artifacts like models, requirements, and test cases. As you set up your process model, you can quickly test your process model by performing dry runs. A dry run can help you test your process model by validating task inputs and generating representative task outputs without actually running the tasks. Dry runs can be helpful for quickly testing your process model and CI pipelines to help make sure they are set up as expected.

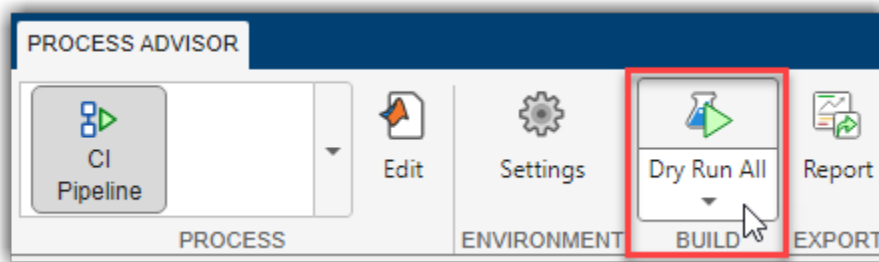
Dry Run Tasks

In the Process Advisor app, you can:

- Dry run a specific task by pointing to the task, opening the options menu (...), and then clicking **Dry Run Task**.
- Dry run each task in the process by clicking **Run All > Dry Run All** in the toolbar.

By default, these dry run buttons appear in the options menu and toolbar. But if you frequently use dry runs, you can set dry runs as the default task execution mode by clicking **Run All > Set Dry Run as Default**. This allows you to:

- Dry run a specific task by pointing to the task and clicking the dry run button .
- Dry run each task in the process by clicking **Dry Run All** directly in the Process Advisor toolbar.





The **Set Dry Run as Default** option is stored in the `DryRunDefaultMode` property in the user settings and only applies to the current MATLAB session. For more information, see `padv.UserSettings`.


















Alternatively, you can dry run tasks programmatically by using the `runprocess` function. Use the `DryRun` argument to perform a dry run. When you programmatically perform a dry run, you can also specify whether the dry run automatically checks out the licenses associated with the tasks, you can specify the `DryRunLicenseCheckout`. For CI systems, dry runs can help you make sure that you have the correct setup and required licenses available on your CI agent and that your pipeline appears as expected. For more information, see `runprocess` and “Tips for Setting Up CI Agents” on page 3-28.

Dry Run Results

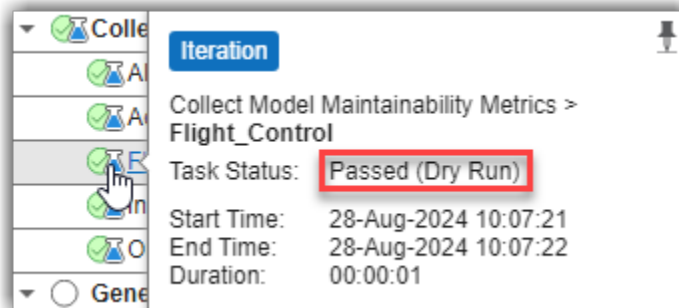
When you dry run a task, the task can validate task inputs and generate representative task outputs.

In the Process Advisor app, the dry run results have a beaker icon  next to the task status. For

example, a passing dry run result shows the passing icon with the beaker icon . If you point to the task results in the **I/O** column, you can see the inputs, placeholder outputs, and dependencies for the task. The results from the dry run are placeholders and are not valid results. Do not use dry run results for anything other than testing your process model and its file management.

Tasks	I/O
<ul style="list-style-type: none">   Collect Model Maintainability Metrics   AHRS_Voter   Actuator_Control   Flight_Control   InnerLoop_Control   OuterLoop_Control 	    

When you point to the status icon next to a task, a pop-up shows details like the task name, status, and duration. If the task was a dry run, the status includes **(Dry Run)**.



Note that Process Advisor and the build system treat the dry run results as normal task run results. If a task has up-to-date dry run results and you re-run the task, the build system automatically skips re-running the task because the dry run results are already up-to-date. If you dry run a task and then want to perform a normal run of the task, you need to clean the task results before trying to re-run the task. To clean the results for a specific task, you can point to that task, open the options menu (...), and click **Clear results and delete outputs**.

Specify Dry Run Functionality for Tasks

Each built-in task has a specialized `dryRun` method to help you evaluate the setup of task inputs and outputs for that task in the process model. For custom tasks, you can either inherit the default dry run behavior or create specialized dry run functionality for your task. Optionally, you can change how your tasks perform dry runs by:

- Overriding the `dryRun` method for class-based tasks
- Specifying the task property `DryRunAction` for function-based tasks
- Changing the default dry run results for tasks in your process model by modifying the `DefaultDryRunResults` property for `padv.ProcessModel`. If a task does not have a dry run functionality defined, the task returns these default dry run results.

Override dryRun Method

To override the dry-run functionality for a class-based custom task, you can override the `dryRun` method. You can use the `dryRun` method to define validation criteria for your task iterations, inputs, and outputs. Inside the `classdef`, in the methods, you can add a `dryRun` function that can perform your custom dry run functionality. In general, the `dryRun` method should use the following method signature:

```
function taskResult = dryRun(obj, input)
    ...
end
```

For example, the following code defines a dry run method that takes the current iteration artifact and checks if the artifact is a model (`sl_model_file`). If the artifact is a model, then the dry run generates placeholder output text files for the task. Otherwise, the task returns a failing task status.

```
function taskResult = dryRun(obj, input)
    taskResult = padv.TaskResult;
    iterationArtifact = input{1};













    if ismember('sl_model_file',iterationArtifact.Type)
        % If input is model, output text file with same name as model
        modelName = iterationArtifact.Alias;
        taskResult.OutputPaths = fullfile(obj.resolvePath(obj.OutputDirectory),...
            modelName+".txt");
    else
        taskResult.Status = padv.TaskStatus.Fail;
        disp('Invalid input. Expected SLX model file.')
    end
end
```

Change Default Dry Run Results

By default, if a task does not have a dry run functionality defined, the task returns the default dry run results specified by the `padv.ProcessModel` property `DefaultDryRunResults`. You can create a different set of default dry run results by creating and using a `padv.TaskResult` object with different property values. For example, to have the default dry run results be failing task results with specific result values in the **Details** column, in your process model you can create a `padv.TaskResult` object and update the value of the `DefaultDryRunResults` property.

```
res = padv.TaskResult;
res.Status = padv.TaskStatus.Fail;
res.ResultValues = struct(...
    "Pass",1,...
    "Warn",2,...
    "Fail",3);
pm.DefaultDryRunResults = res;
```

When you dry run a task that does not already have a defined dry run behavior, the task uses the specified default dry run results.

Tasks	I/O	Details
▼  MyCustomTask		✓ 5 △ 10 ✗ 15
 AHRs_Voter		✓ 1 △ 2 ✗ 3
 Actuator_Control		✓ 1 △ 2 ✗ 3
 Flight_Control		✓ 1 △ 2 ✗ 3
 InnerLoop_Control		✓ 1 △ 2 ✗ 3
 OuterLoop_Control		✓ 1 △ 2 ✗ 3

See Also

`padv.ProcessModel` | `padv.Task` | `padv.TaskResult`

Related Examples

- “Overview of Process Model” on page 2-9
- “Add Tasks to Process” on page 2-13
- “Create Custom Tasks” on page 2-28
- “Test Tasks and Queries” on page 2-63

Troubleshoot Missing Tasks, Artifacts, and Dependencies

When you use CI/CD Automation for Simulink Check, the support package creates a digital thread to capture the attributes and unique identifiers of certain artifacts in your project. The digital thread is a set of metadata information about artifacts in a project, artifact structures, and the traceability relationships between artifacts. The Process Advisor app and build system monitor and analyze the digital thread to identify artifacts, detect changes to project files, generate task iterations, and identify outdated task results. The digital thread is cached in a database stored in `derived > artifacts.dmr` in the project. The digital thread only tracks changes to specific types of artifacts. For information on supported artifact types, see “Valid Artifact Types” on page 2-25.

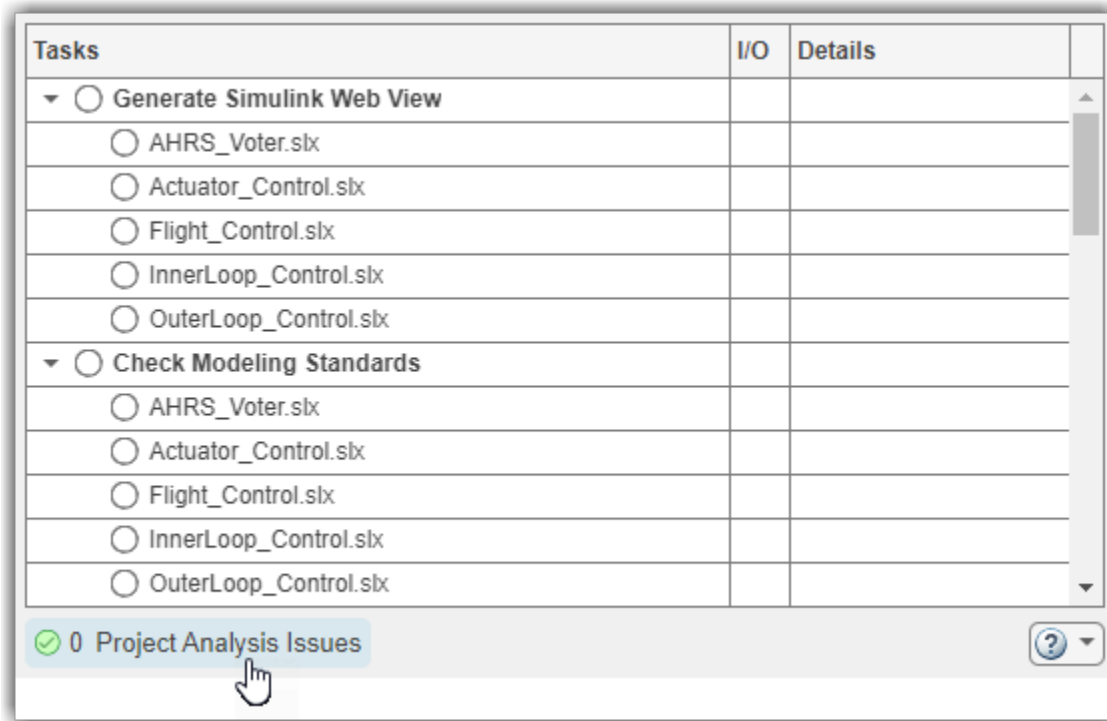
Artifact Issues

Before you begin troubleshooting Process Advisor or the build system:



- Check if your artifacts are saved in the project. Project references are supported starting in R2023a. You can use files outside your project as inputs to tasks, but the files appear as **Untracked** because you cannot track changes to files outside the project. If you make a change to an untracked file, the build system does not mark the task as outdated.
- Artifacts are on the MATLAB search path before you open the Process Advisor app.
- You used the Process Advisor app or build system to run your tasks and to collect task results.
- Artifacts are not saved to a prohibited output folder. Prohibited output folders include the simulation cache, project resources folder, and `.SimulinkProject`.
- You have a compiler configured. You should use the same compiler that you use in the target development environment. If you only have the MinGW® compiler installed on your system, the `mex` command automatically chooses MinGW.
- Make sure your tests are testing a model or an atomic subsystem, Stateflow chart, MATLAB function, or subsystem reference.

Project Analysis Issues

At the bottom of the Process Advisor app is a **Project Analysis Issues** pane. After Process Advisor analyzes the project, the **Project Analysis Issues** shows errors or warnings that the artifact analysis generated.



1 Investigate project analysis issues in the project by clicking on **Project Analysis Issues**.

- An error  indicates that Process Advisor might not have been able to properly analyze artifacts, trace artifacts, or identify outdated results, so the information shown by Process Advisor might be incomplete.
- A warning  indicates that Process Advisor does not support that specific artifact, modeling construct, or relationship.

2 Fix the issues listed in the **Project Analysis Issues** pane to make sure the app can fully analyze the project, generate the expected task iterations, and detect outdated results.

When there are issues with an artifact, check that the artifact does not use the following unsupported modeling constructs. The digital thread does a static analysis of your project. Certain modeling constructs dynamically add unsaved information or ambiguous relationships. The digital thread does not detect these changes in the project and the changes do not cause related task results to become outdated.

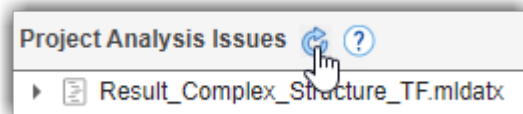
Affected Artifact	Unsupported Construct
Library	Library forwarding table
	Self-modifiable masks
Model	Saved in release R2012a or earlier
	Model loading callbacks
	Model shadowing
Test case	MATLAB-based Simulink test
Test file	Test-file level callbacks

Affected Artifact	Unsupported Construct
Test suite	Test-suite level callbacks

Note To test libraries with Process Advisor, specify function interfaces for each of your library blocks and use the library-based code generation workflow. For more information, see “Library-Based Code Generation for Reusable Library Subsystems” (Embedded Coder).

Make sure you only use the library blocks in the model context that you verified. When you test the model, you can use coverage filters to exclude the library blocks that you already tested.

- Click the refresh button in the pane to refresh the list of project analysis issues.



- If you want to filter out certain types of issues, you can get the project settings, `padv.ProjectSettings.get()`, and add issue IDs to the `FilteredDigitalThreadMessages` property value.

To get a list of the issue messages and issue IDs, use the function `getArtifactIssues`.

```
metric_engine = metric.Engine();
issues = getArtifactIssues(metric_engine)
issuesMessages = issues.IssueMessage
issueIDs = issues.IssueId
```

Suppose that you want to filter out the issue message associated with the issue ID "alm:artifact_service:CannotResolveElement". You can use the method `addFilteredDigitalThreadMessages` to add the issue message to the list of filtered messages:

```
ps = padv.ProjectSettings.get();
ps.addFilteredDigitalThreadMessages(...
"alm:artifact_service:CannotResolveElement");
```

For more information, see `padv.ProjectSettings`.

Limitations on Incremental Build

There are changes that incremental build does not detect. Tasks depending on those changes will remain up-to-date and will not execute with **Run All**. If incremental build does not detect changes to a file that a task depends on, the file is an *undetected dependency*. For example, if you have a model that uses a referenced global workspace variable and you make a change to the variable, the task results associated with the model will not become outdated.


The table in this section lists the known untracked dependencies.

- The **Artifact** column lists the artifacts with known untracked dependencies.
- The **Undetected Dependency** column lists the files that incremental build does not detect changes to. Changes to these files do not cause tasks associated with the artifact to become outdated.

Artifact	Undetected Dependency
Model	Model callbacks
	Referenced global workspace variables
	Global enumeration definitions
	Externally-saved model workspace variables (if auto-initialized)
	Data or functions referenced in masks or callbacks inside the model
	Known dependencies specified in the model reference rebuild options of a configuration set
	Simulation inputs and simulation outputs specified in model configuration sets
	Signal Editor scenarios
	C code referenced in C Caller blocks
	Code inside SIL (software-in-the-loop) blocks
	Files associated with S-Functions
	Code replacement libraries
	Custom code
System Composer profiles or stereotypes	
Test case	MATLAB code in: <ul style="list-style-type: none"> • Pre-load, post-load, clean-up, and assessment callbacks • Custom criteria
	External configurations
	MATLAB test files

If possible, use a Simulink Data Dictionary file instead of referenced global workspace variables or global enumeration definitions. The digital thread tracks changes to data dictionaries.

If you do not want the build system or the Process Advisor app to run incremental builds, you can disable incremental builds for a project. For more information, see “Specify Settings for Process Advisor and Build System” on page 1-16. You can also force up-to-date tasks to execute by using one of these approaches:

- In the Process Advisor app, either point to a task and click the run button  or click **Run All > Force Run All**.
- For the runprocess function, specify Force as true.

Other Limitations

There are known limitations in the Process Advisor app and build system:

- Process Advisor only shows results for tasks that you ran using Process Advisor and the build system.
- If a top model and at least one referenced model have unsaved changes, the Process Advisor is unable to save the top model and generates the error: The following files were not able to be saved: *<Path to top model>*

- For the **Check Coding Standards** task, if you specify `PsAccessEnable` as `true`, make sure you also specify values for the other Polyspace Access™ Configuration Options. For information, see “Upload Results to Polyspace Access”.

If you do not specify the other required configuration options, the task returns an error: Task 'padv.builtin.task.AnalyzeModelCode' threw unhandled exception 'Invalid argument at position 2. Value must not be empty.'

- Before you use the pipeline generator, make sure that all of the products used by your pipeline are licensed and installed. If a product is not licensed or installed, the pipeline generator returns an error message: Error using + Not enough input arguments. Error in padv.pipeline.internal.gitlab.PipelineGenerator/createGitlabYMLContent (line 166) gitlabPipelineFullPath = obj.GitlabOptions.PipelineDirRelPath + '###' + gitlabPipeline.Name;.
- Your task results can unexpectedly become outdated if you use one of the following queries as an input query and specify non-empty values for `IncludeLabel`, `ExcludeLabel`, `IncludePath`, or `ExcludePath`:
 - `padv.builtin.query.FindCodeForModel`
 - `padv.builtin.query.FindDesignModels`
 - `padv.builtin.query.FindRequirementsForModel`
 - `padv.builtin.query.FindTestCasesForModel`
 - `padv.builtin.query.FindTopModels`
 - `padv.builtin.query.FindUnits`

If you see this behavior, consider using a different query, like `padv.builtin.query.FindArtifacts`, instead.

Resolve Path Issues

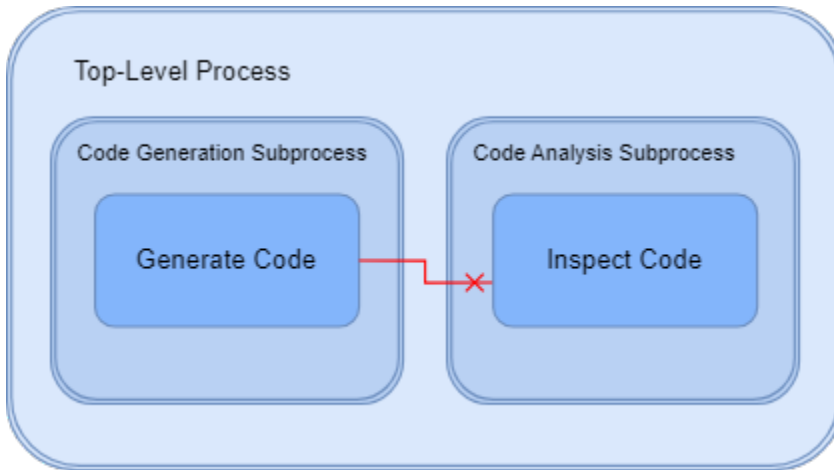
If an artifact is not on the MATLAB search path, add the artifact to your project, then close and re-open the project. When you re-open the project, the MATLAB search path reflects the updated search path.

Handling Invalid Dependencies

Suppose you have one subprocess that contains your code generation tasks and another subprocess that contains your code analysis tasks.

```
spCodeGen = pm.addSubprocess("Code Generation Tasks");  
spCodeAnalysis = pm.addSubprocess("Code Analysis Tasks");
```

Your code analysis tasks need access to the generated code, but the tasks themselves cannot directly depend on the code generation task because that relationship would cross the subprocess boundary.



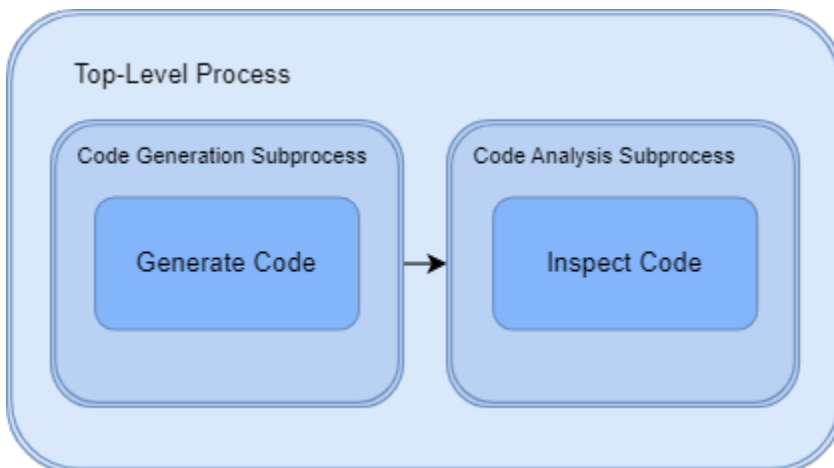
If you try to have a code analysis task in one subprocess depend on a code generation task in another subprocess, Process Advisor generates an error like: Invalid dependency between Task 'padv.builtin.task.RunCodeInspection' and 'padv.builtin.task.GenerateCode'. Make sure 'padv.builtin.task.GenerateCode' exists in the current process and that the dependency does not cross any subprocess boundaries.

To pass the generated code from your code generation subprocess to your code analysis subprocess, you can:

- Update code analysis tasks, like RunCodeInspection, to find and use the generated model code as an input to the task using the built-in query `padv.builtin.query.FindCodeForModel`
- Specify that the code analysis subprocess depends on the code generation subprocess

```
% Update code analysis tasks to find
% and use model code as an input to the task
slciTask = spCodeAnalysis.addTask(...
    padv.builtin.task.RunCodeInspection(...
        InputQueries=padv.builtin.query.FindCodeForModel));

% Code Analysis Subprocess depends on Code Generation Subprocess
spCodeAnalysis.dependsOn(spCodeGen);
```



Analyze Project From Scratch

If you experience unexpected project analysis issues, you can clear the current project analysis and analyze your project from scratch by calling the function `padv.util.forceReanalyzeProject`:

```
padv.util.forceReanalyzeProject()
```

The function forces a reanalysis of the current project by creating backups of the existing artifact database (`artifacts.dmr`), clearing the existing project analysis, and reanalyzing the project. The function also logs project analysis events, which can help with troubleshooting persistent project analysis issues. Note that when you run the function, the function closes and reopens the project.

For more information, see the utility function `padv.util.forceReanalyzeProject`.

Note You should only use the function `padv.util.forceReanalyzeProject` when there are unexpected project analysis issues. When you clear the existing project analysis file, you might permanently lose important information, including the UUIDs that the digital thread assigned to artifacts in your project. Reanalyzing a project might take some time to complete. The `artifacts.dmr` file might be used by other project users and if you use other tools that use the digital thread, you might need to re-run the metrics in those tools.

For general task and result cleanup, use `runprocess` instead. The `runprocess` function has name-value arguments, `Clean` and `DeleteOutputs`, that you can use to clean task results and delete task outputs. For information, see `runprocess`.

See Also

`generateProcessTasks` | `padv.util.forceReanalyzeProject` | `runprocess`

Related Examples

- “Resolve Missing Artifacts, Links, and Results”

Integrate Process into CI

Approaches to Running Processes in CI

With the support package CI/CD Automation for Simulink Check, you can define a process for your team and set up your CI system to automatically run that process when you push code changes to your repository, create a pull request, or perform other pipeline triggering events. By automatically running your process, you can help your team find and fix problems in the software and improve software quality. For more information on CI workflows and benefits, see “Develop and Integrate Software with Continuous Integration”.

Before You Integrate

Your process model file defines the pipeline of tasks that the `runprocess` function runs in CI. If you do not already have a process model, the support package includes process model templates that you can use to get started. For more information, see “Automate and Run Tasks with Process Advisor” on page 1-2.

Before you try to run your process as part of an automated pipeline of tasks in CI, you need to connect your CI platform, remote repository, and project.

- 1 Choose a CI platform to run MATLAB. MATLAB integrates with common CI platforms like GitHub®, GitLab®, Jenkins®, and other CI platforms.
- 2 Create a remote repository for your project. Many platforms, like GitHub and GitLab, provide source-controlled remote repositories as part of their platform. For other CI platforms, like Jenkins, you need to host your remote repository on another platform. See the documentation for your chosen CI platform to identify how you want to set up your remote repository.
- 3 Set up a CI agent. Your CI agent machine is responsible for running MATLAB and communicating the results back to your chosen CI platform. Depending on the CI platform, you can set up the platform to run MATLAB on your own, self-hosted machine or in the cloud. Make sure that your CI agent can run MATLAB and that you install the support package and any other products required by your process. For more information, see “Tips for Setting Up CI Agents” on page 3-28.
- 4 Connect your project, remote repository, and CI platform. On the **Project** tab, in the **Source Control** section, click **Remote** and specify the URL for your remote repository. For more information, see “Use Source Control with MATLAB Projects”.
- 5 Make sure that your process model file is available on the MATLAB path for your CI agent. As a best practice, keep your process model file in the project root folder and add the process model file to the project.

Depending on your CI platform, you have different options for how to configure and run your process in CI.

GitHub

Basic Integration

You can manually author a GitHub Actions workflow file that opens your project and runs your process as part of a workflow. In the file, you can define a sequence of steps that checks out your repository code, opens your project using the `openProject` function, and runs your process using the `runprocess` function. To access an example workflow file and project, enter `processAdvisorGitHubExampleStart` at the MATLAB command line.

Recommended Integration

For a more robust and customizable CI integration, you can generate GitHub Actions workflow file for your process by using the pipeline generator in the support package. When you use the generated files in the workflow that you define in the `.github/workflows` directory of your repository, your project can create a pipeline of tasks for your process in CI. You can create pipelines that separate your tasks into different jobs and use other custom pipeline behaviors.

For more information, see “Integrate Process into GitHub” on page 3-5.

GitLab

Basic Integration

You can modify the MATLAB YAML template to run the `openProject` and `runprocess` functions as commands in GitLab. For more information, see Use MATLAB with GitLab CI/CD.

Recommended Integration

For a more robust and customizable CI integration, use the Process Advisor YAML template as your pipeline YAML file (`.gitlab-ci.yml`). After you add the template to your project and perform a one-time setup, your project can automatically create pipelines with different jobs for each task in your process in CI. You can reconfigure the template to create pipelines that separate your tasks into different jobs and use other custom pipeline behaviors. The template uses the pipeline generator to analyze your project and process model to automatically generate the necessary pipeline files for you, so that you do not need to manually update those files when you make changes to the tasks and artifacts in your project. Inside the `script` section of the template, you specify the pipeline generation options.

For more information, see “Integrate Process into GitLab” on page 3-8.

Jenkins

Basic Integration

You can install the MATLAB plugin on your Jenkins agent and use the **Run MATLAB Command** build step to open your project and run your process with the `openProject` and `runprocess` functions. For more information, see the plugin on Jenkins Plugins Index.

Recommended Integration

For a more robust and customizable CI integration, use the Process Advisor Jenkinsfile template. After you add the template to your project and perform a one-time setup, your project can automatically create pipelines with different jobs for each task in your process in CI. You can reconfigure the template to create pipelines that separate your tasks into different jobs and use other custom pipeline behaviors. The template uses the pipeline generator to analyze your project and process model and automatically generate the necessary pipeline files for you, so that you do not need to manually update those files when you make changes to the tasks and artifacts in your project. Inside the Pipeline Generation stage of the template, you specify the pipeline generation options.

For more information, see “Integrate Process into Jenkins” on page 3-14.

Other Platforms

For other platforms, you can use the `matlab` command with the `-batch` option in your CI system. You can use `matlab -batch` to run MATLAB code, including the `openProject` and `runprocess` functions, noninteractively. For example, `matlab -batch "openProject(pwd);runprocess();"` starts MATLAB noninteractively, opens the project in the current working directory, and runs each of the tasks in the pipeline defined by the available process model file (`processmodel.p` or `processmodel.m`). MATLAB terminates automatically with the exit code `0` if the specified code executes successfully without generating an error. Otherwise, MATLAB terminates with a nonzero exit code.

For more information, see “Continuous Integration with MATLAB on CI Platforms”.

See Also

`matlab` | `openProject` | `padv.pipeline.generatePipeline` | `runprocess`

Related Examples

- “How Pipeline Generation Works” on page 3-21
- “Integrate Process into GitHub” on page 3-5
- “Integrate Process into GitLab” on page 3-8
- “Integrate Process into Jenkins” on page 3-14
- “Tips for Setting Up CI Agents” on page 3-28

Integrate Process into GitHub

You can define a process for your team and set up your CI system to run the tasks in that process as a pipeline in CI using the CI/CD Automation for Simulink Check support package.

In this example, you connect a project to GitHub and generate a GitHub Actions workflow file for the project and its process model by using the pipeline generator. You can specify the pipeline generator options to create pipelines that separate tasks into different jobs and use other custom pipeline behaviors.

This example shows the recommended way to use your process in GitHub. Alternatively, you can manually author a GitHub Actions workflow file that opens your project and runs your process. For more information, “Approaches to Running Processes in CI” on page 3-2.

Set Up GitHub Project and Runner

To set up the CI system, you need to set up a source-controlled remote repository where you store your project and a CI agent machine that can run your pipeline on that repository. For this example, you can use GitHub as both your remote repository and CI system, and then create a self-hosted GitHub runner to run your pipelines.

- 1 In GitHub, create a private GitHub repository. For information, see Quickstart for repositories in the GitHub documentation. Make sure GitHub Actions is enabled for your repository.
- 2 Create a self-hosted runner. See the GitHub documentation for Adding self-hosted runners.
- 3 Install MATLAB, Simulink, Simulink Check, the CI/CD Automation for Simulink Check support package, and any other products that your process requires on the machine that your GitHub runner is running on. Make sure that your GitHub runner machine can access and run MATLAB before you continue.

For information on licensing considerations, Docker® containers, and virtual displays, see “Tips for Setting Up CI Agents” on page 3-28.

Connect MATLAB Project to GitHub

You need to connect your MATLAB project to your remote repository so that you can push your changes to the remote GitHub repository and allow GitHub to automate a CI pipeline for the project.

- 1 Open a project in MATLAB. For this example, open an example project that uses the process defined by an example process model.

```
processAdvisorExampleStart
```

The process model, `processmodel.m`, is at the root of the project and defines a process with common model-based design tasks. You can use the Process Advisor app to run the tasks in the process on your local machine. For information on how to create and customize a process model for your development and verification workflow, see “Customize Your Process Model”.

- 2 On the **Project** tab, in the **Source Control** section, click **Remote** and specify the URL for the remote origin in GitHub where your repository is hosted. For example, `https://github.com/user/repo.git`.

The example project is already set up to use local Git™ source control. For information on how to use source control with your projects, see “Use Source Control with MATLAB Projects”.

Generate Pipeline Configuration File

You generate the GitHub Actions workflow file by using the pipeline generator.

- 1 In MATLAB, configure the pipeline generation options by creating a `padv.pipeline.GitHubOptions` object and specifying the location of your MATLAB installation for your runner.

The `padv.pipeline.GitHubOptions` object stores the options for the pipeline generator. You can modify the other properties of the object to customize how the pipeline generator creates your pipeline configuration file. For example, you can create a `padv.pipeline.GitHubOptions` object for a GitHub runner that uses a MATLAB installation at `/opt/matlab/r2023a`.

```
GitHubOptions = padv.pipeline.GitHubOptions
GitHubOptions.MatlabInstallationLocation = "/opt/matlab/r2023a";
```

By default, `GitHubOptions` specifies a `SingleStage` pipeline architecture that runs all the tasks in the process within a single stage in CI. To change the number of stages or the grouping of tasks in the CI pipeline, specify the `PipelineArchitecture` property of your `padv.pipeline.GitHubOptions` object.

- 2 Generate a pipeline configuration file for your project by calling the `padv.pipeline.generatePipeline` function on your `padv.pipeline.GitHubOptions` object.

```
padv.pipeline.generatePipeline(GitHubOptions)
```

By default, the generated pipeline configuration file is named `simulink_pipeline.yml` and is located under the project root, in the subfolder **derived > pipeline**.

The generated pipeline configuration file uses the following GitHub Actions:

- `checkout@v3`
- `cache@v3`
- `upload-artifact@v3`
- `download-artifact@v3`

Use Pipeline Configuration File in GitHub Actions Workflow

To use the generated pipeline configuration file in your GitHub repository, you need to create a workflow and update the workflow file.

- 1 In GitHub, create a GitHub Actions workflow by creating a directory `.github/workflows` and creating a new YAML file `github-actions-demo.yml`. See <https://docs.github.com/en/actions/quickstart#creating-your-first-workflow>.
- 2 In MATLAB, open your generated pipeline configuration file and copy the file contents.
- 3 In GitHub, paste the contents of `simulink_pipeline.yml` inside the `github-actions-demo.yml` file.
- 4 Check the new `github-actions-demo.yml` file into your repository by committing the changes and creating a pull request.

After you commit your changes, GitHub automatically runs the workflow file, `github-actions-demo.yml`. You can see your process running when you click on the **Actions** tab. For information on

the GitHub workflow results, see <https://docs.github.com/en/actions/quickstart#viewing-your-workflow-results>.

See Also

`padv.pipeline.generatePipeline` | `padv.pipeline.GitHubOptions`

Related Examples

- “Approaches to Running Processes in CI” on page 3-2
- “How Pipeline Generation Works” on page 3-21
- “Tips for Setting Up CI Agents” on page 3-28

Integrate Process into GitLab

You can define a process for your team and set up your continuous integration (CI) system to automatically run the tasks in that process as a pipeline by using the CI/CD Automation for Simulink Check support package. The support package includes a GitLab template that defines CI jobs for each task in your process. You can reconfigure the template file to separate your tasks into different jobs and customize other pipeline behaviors. The template file uses the pipeline generator to automatically generate pipelines for you, so that you do not need to manually update CI/CD configuration files when you make changes to your project or processes.

This example shows the how to:

- Set up a GitLab project and CI agent.
- Connect a MATLAB project to GitLab.
- Perform a one-time setup of a GitLab template file to work with your CI setup.
- Push changes to source control and inspect the automatically generated pipeline.

This example shows the recommended way to use your process in GitLab. Alternatively, you can manually author a pipeline configuration file that opens your project and runs your process as part of your build. For more information, “Approaches to Running Processes in CI” on page 3-2.

Set Up GitLab Project and Runner

Set up a source-controlled remote repository where you store your project and a CI agent machine that can run your pipeline on that repository. For this example, you can use GitLab as both your remote repository and CI system, and then create a GitLab Runner to run your pipelines.

- 1** In GitLab, set up a remote repository by creating a new blank project. See the GitLab documentation for Create a project.
- 2** Install, register, and start a GitLab Runner on a machine. The GitLab Runner application allows a machine to act as a CI agent in GitLab. If you assign a tag to your Runner, make note of the tag name. See the GitLab documentation for Install GitLab Runner.
- 3** Install MATLAB, Simulink, Simulink Check, the CI/CD Automation for Simulink Check support package, and any other products that your process requires on the machine that has your GitLab Runner. Make sure that your GitLab Runner machine can access and run MATLAB before you continue.

For information on licensing considerations, Docker containers, and virtual displays, see “Tips for Setting Up CI Agents” on page 3-28.

Note To run CI jobs in parallel, you must either:

- Have multiple runners available.
 - Configure your runner to run multiple jobs concurrently by specifying the concurrent setting. See the GitLab documentation for Advanced configuration.
-

Connect MATLAB Project to GitLab

Connect your MATLAB project to your remote repository so that you can push your changes to the remote GitLab repository and allow GitLab to automate a CI pipeline for the project.

- 1 Open a project in MATLAB. For this example, open an example project that uses an example process model.

```
processAdvisorExampleStart
```

The process model, `processmodel.m`, is at the root of the project and defines a process with common model-based design tasks. You can use the Process Advisor app to run the tasks in the process on your local machine. You can copy the default process model template into a project by entering `createprocess(Template = "default")` at the command line. For information on how to customize a template process model for your development and verification workflow, see “Customize Your Process Model”.

- 2 In MATLAB, on the **Project** tab, in the **Source Control** section, click **Remote** and specify the URL for the remote origin in GitLab where your repository is hosted. For example, `https://gitlab.com/gitlab-org/gitlab.git`.

The example project is already set up to use local Git source control. For information on how to use source control with your projects, see “Use Source Control with MATLAB Projects”.

Configure Template to use GitLab Runner

In GitLab, you define your CI pipelines by using a CI/CD configuration file, typically named `.gitlab-ci.yml`, in your project root. The support package includes a GitLab template file that you can reconfigure and then use to automatically generate pipelines.

- 1 Copy the GitLab template into your project folder. The GitLab template file is generic and can work with any project. In MATLAB, change your current folder to your project root and enter:

```
GitLabTemplate = fullfile(...
matlabshared.supportpkg.getSupportPackageRoot,...
"toolbox","padv","samples",".gitlab-ci-pipeline-gen.yml");
```

```
copyfile(GitLabTemplate,".gitlab-ci.yml")
```

- 2 In the **Project** pane, add the template file, `.gitlab-ci.yml`, to your project. The template file contains a CI pipeline definition for GitLab.
- 3 Open and inspect the template file. The file uses GitLab CI/CD YAML syntax to define a parent pipeline that can generate and execute pipelines for you.
- 4 Reconfigure the template to work for your CI setup. In the template, find and replace instances of `padv_demo_ci` with the tag name of the GitLab Runner that you want to use.

For example, if your GitLab Runner has the tag name `high_memory`, you specify that tag in the `tags` section and in the pipeline generation options object `padv.pipeline.GitLabOptions`.

```
variables:
  MATLAB_LOG_FILE: "MATLAB_Log_Output.txt"
  # Set the value of below variable to 'none', if you do not wish
  # to fetch the git submodules
  GIT_SUBMODULE_STRATEGY: recursive

stages:
  - SimulinkPipelineGeneration
  - SimulinkPipelineExecution

# Do not change the name of the jobs in this pipeline
SimulinkPipelineGeneration:

  stage: SimulinkPipelineGeneration

  tags:
    - high_memory

  script:
    # Open the project and generate the pipeline using
    # appropriate options in project root
    - >
      matlab
      -nodesktop
      -logfile "$MATLAB_LOG_FILE"
      -batch "
cp = openProject(pwd);
padv.pipeline.generatePipeline(
padv.pipeline.GitLabOptions(
PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask,
Tags = 'high_memory',
GeneratedVMLFileName = 'simulink_pipeline.yml',
GeneratedPipelineDirectory = fullfile('derived','pipeline')));
```

GitLab Runner tag

Pipeline Generator command

The template file can then generate a GitLab pipeline with stages for each task in your process the next time that you push your changes to your remote repository. Optionally, you can further customize the template file to change how the pipeline generator organizes and executes the pipeline. You can dry run your tasks, separate your tasks into different jobs, and specify other options by using the `padv.pipeline.GitLabOptions` object in the template.

Make Optional Customizations

Optionally, you can reconfigure the template file to customize how the pipeline generator organizes and executes the pipeline. To customize the pipeline generator options, modify the property values of the `padv.pipeline.GitLabOptions` object in the template.

For example, suppose that you want to:

- Dry run tasks to quickly validate task inputs and generate representative outputs without performing the full task operation.
- Perform license checkouts during the dry runs to make sure that your GitLab Runner has access to the required products.
- Separate tasks into different jobs.

To change how the template file generates the pipeline, you can modify the `padv.pipeline.GitLabOptions` in the script section.

```
script:
  # Open the project and generate the pipeline using
  # appropriate options in project root
  - >
    matlab
    -nodesktop
    -logfile "$MATLAB_LOG_FILE"
```

```

-batch "
cp = openProject(pwd);
rpo = padv.pipeline.RunProcessOptions;
rpo.DryRun = true;
rpo.DryRunLicenseCheckout = true;
padv.pipeline.generatePipeline(
padv.pipeline.GitLabOptions(
PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask,
RunprocessCommandOptions = rpo,
Tags = 'padv_demo_ci',
GeneratedYMLFileName = 'simulink_pipeline.yml',
GeneratedPipelineDirectory = fullfile('derived','pipeline')));
"

```

This example code creates a `padv.pipeline.RunProcessOptions` object, `rpo`, that customizes the behavior of the `runprocess` function in CI. This code specifies the `runprocess` arguments `DryRun` and `DryRunLicenseCheckout` as `true`, updates the `padv.pipeline.GitLabOptions` object to use the `PipelineArchitecture SerialStagesGroupPerTask`, and uses the `RunprocessCommandOptions` specified by `rpo`. For more information, see “How Pipeline Generation Works” on page 3-21.

If you modify other parts of the template file, make sure that your changes use valid GitLab CI/CD YAML syntax. For more information, see the GitLab documentation for CI/CD YAML syntax reference.

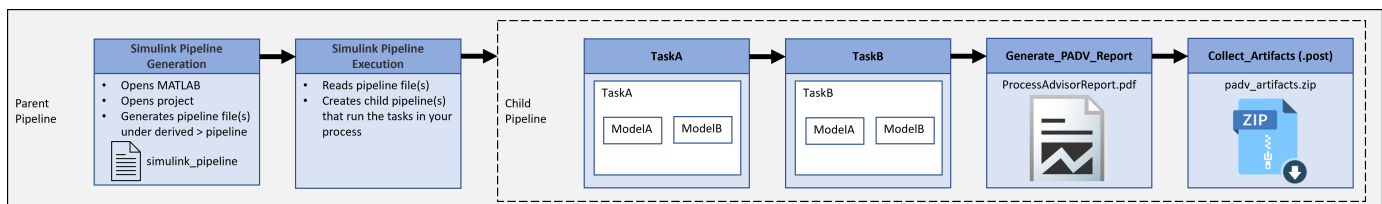
Generate Pipeline in GitLab

Commit and push the MATLAB project to your GitLab repository by using the **Commit** and **Push** buttons in the **Source Control** section of the **Project** tab. By default, GitLab uses `.gitlab-ci.yml` as the CI/CD configuration file to automatically create pipelines when triggered.

Each time you submit changes to this remote repository, GitLab generates and executes a custom pipeline for your project, process, and pipeline generation options. You do not need to update the `.gitlab-ci.yml` file when you make changes to your projects or process model. The pipeline generator automatically generates up-to-date pipelines by using the latest project and process model. You only need to update the `.gitlab-ci.yml` file if you want to change how the pipeline generator organizes and executes the pipeline.

In GitLab, your pipeline contains two upstream jobs:

- **SimulinkPipelineGeneration** — Generates a child pipeline file.
- **SimulinkPipelineExecution** — Executes the child pipeline file. By default, the child pipeline contains:
 - Jobs for your process, organized by the `PipelineArchitecture` property specified in `padv.pipeline.GitLabOptions`.
 - The `Generate_PADV_Report` job, which generates a Process Advisor build report.
 - The `Collect_Artifacts` job, which collects build artifacts.



Optional Customizations

You can reconfigure the template file to customize how the pipeline generator organizes and executes the pipeline. To customize the pipeline generator options, you modify the property values of the `padv.pipeline.GitLabOptions` object in the template.

For example, suppose that you want to:

- Dry run tasks to quickly validate task inputs and generate representative outputs without performing the full task operation.
- Perform license checkouts during the dry runs to make sure that your GitLab Runner has access to the required products.
- Separate tasks into different jobs.

To change how the template file generates the pipeline, you can modify the `padv.pipeline.GitLabOptions` in the `script` section.

```
script:
# Open the project and generate the pipeline using
# appropriate options in project root
- >
  matlab
  -nodesktop
  -logfile "$MATLAB_LOG_FILE"
  -batch "
cp = openProject(pwd);
rpo = padv.pipeline.RunProcessOptions;
rpo.DryRun = true;
rpo.DryRunLicenseCheckout = true;
padv.pipeline.generatePipeline(
padv.pipeline.GitLabOptions(
PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask,
RunprocessCommandOptions = rpo,
Tags = 'padv_demo_ci',
GeneratedYMLFileName = 'simulink_pipeline.yml',
GeneratedPipelineDirectory = fullfile('derived','pipeline')));
"
```

This example code creates a `padv.pipeline.RunProcessOptions` object, `rpo`, for customizing the behavior of the `runprocess` function in CI. In this case, specifying the `runprocess` arguments `DryRun` and `DryRunLicenseCheckout` as `true`. The code updates the `padv.pipeline.GitLabOptions` object to use the `PipelineArchitecture` `SerialStagesGroupPerTask` and use the `RunprocessCommandOptions` specified by `rpo`. For more information, see “How Pipeline Generation Works” on page 3-21.

If you modify other parts of the template file, make sure that your changes use valid GitLab CI/CD YAML syntax. For more information, see <https://docs.gitlab.com/ee/ci/yaml/index.html>.

See Also

`padv.pipeline.generatePipeline` | `padv.pipeline.GitLabOptions`

Related Examples

- “Approaches to Running Processes in CI” on page 3-2

- “How Pipeline Generation Works” on page 3-21
- “Tips for Setting Up CI Agents” on page 3-28

Integrate Process into Jenkins

You can define a development and verification process for your team and run that process as a pipeline in CI using the CI/CD Automation for Simulink Check support package. For Jenkins, you configure your pipeline by using a `Jenkinsfile` that you store in your project. The `Jenkinsfile` can configure different parts of your CI/CD jobs including the stages of the job, the label for the Jenkins agent that executes the pipeline, the script that the agent executes, and artifacts you want to attach to a successful job. The support package provides a template `Jenkinsfile` that you can reconfigure and use to run your process. The template file uses the pipeline generator to automatically generate pipelines for you, so that you do not need to manually update pipeline configuration files when you make changes to the tasks and artifacts in your project.

This example shows the recommended way to use your process in Jenkins. Alternatively, you can run your process as a build step. For more information, “Approaches to Running Processes in CI” on page 3-2.

Set Up Jenkins

- 1 Install Jenkins. See the Jenkins documentation for Installing Jenkins.
- 2 Install the following plugins for Jenkins:
 - MATLAB Plugin for Jenkins. See MATLAB plugin on Jenkins Plugins Index.
 - Jenkins Core Plugin, which allows pipelines to archive artifacts using the `archiveArtifacts` step. See the Jenkins documentation for `archiveArtifacts`.
 - JUnit Plugin, which allows Jenkins to show test failures and trends directly in the user interface. See JUnit plugin on Jenkins Plugins Index.
 - Job Cacher Plugin, which allows Jenkins to store caches. See Job Cacher plugin on Jenkins Plugins Index.

The pipeline generator requires the `skipSave` parameter that was introduced in plugin version `399.v12d4fa_dd3db_d`. Pipeline generation was tested using plugin version `481.v15f51ca_4c6b_7`.

- 3 Install MATLAB, Simulink, Simulink Check, the CI/CD Automation for Simulink Check support package, and any other products that your process requires on your Jenkins agent. Make sure that your Jenkins agent machine can access and run MATLAB before you continue.
- 4 Configure at least 3 executors on your Jenkins instance. Executors control the number of concurrent tasks or builds Jenkins can run. The number of required executors depends on the pipeline architecture that you select. You must have at least 3 executors configured and available to load, generate, and execute your pipeline using the pipeline generator. If you are using the `IndependentModelPipelines` pipeline architecture, you need at least 3 executors, plus an additional executor for each model in your project. For more information, see `PipelineArchitecture`. For information on defining Jenkins executors, see `Managing Nodes`.
- 5 Create a new Jenkins pipeline project, but leave your `Jenkinsfile` pipeline definition empty for now. See the Jenkins documentation for Getting started with Pipeline. Instead of manually writing a `Jenkinsfile`, you reconfigure the template `Jenkinsfile`, add that file to your MATLAB project, and use that file to define your pipelines as shown in “Configure and Use Jenkinsfile Template” on page 3-15.

For information on licensing considerations, Docker containers, and virtual displays, see “Tips for Setting Up CI Agents” on page 3-28.

Connect Jenkins Project to Repository

To set up your CI system, you need to set up a source-controlled remote repository where you store your MATLAB project and you need to connect that repository to your Jenkins project.

For this example, you can set up a GitLab repository and connect that repository to Jenkins.

- 1 Set up a remote GitLab repository by creating a new blank project. See the GitLab documentation for Create a project.
- 2 Connect your MATLAB project to the remote repository.

For this example, you can open the Process Advisor example project `processAdvisorExampleStart` and, on the **Project** tab, in the **Source Control** section, click **Remote** to specify the URL for the remote origin in GitLab where your repository is hosted. For example, `https://gitlab.com/gitlab-org/gitlab.git`.

The process model, `processmodel.m`, is at the root of the project and defines a process with common model-based design tasks. You can use the Process Advisor app to run the tasks in the process on your local machine. You can copy the default process model template into a project by entering `createprocess(Template = "default")` at the command line. For information on how to customize a template process model for your development and verification workflow, see "Customize Your Process Model".

- 3 Configure GitLab integration with Jenkins. See GitLab documentation for Jenkins integration.

Configure and Use Jenkinsfile Template

For Jenkins, you can define your CI pipelines by using a Jenkinsfile. The support package includes a Jenkinsfile template that you can reconfigure and then use to automatically generate pipelines.

- 1 In your MATLAB project, change your current folder to your project root and copy the template Jenkinsfile into your project. The template Jenkinsfile is generic and can work with any project.

```
exampleJenkinsfile = fullfile(...
matlabshared.supportpkg.getSupportPackageRoot,...
"toolbox", "padv", "samples", "Jenkinsfile_pipeline_gen");

copyfile(exampleJenkinsfile, "Jenkinsfile")
```

- 2 Open and inspect the template Jenkinsfile in your project. The file defines a pipeline that checks out code from a specified Git repository, specifies MATLAB environment information, and then uses MATLAB to generate and execute a pipeline file for your specific project and process. The template uses the pipeline generator function, `padv.pipeline.generatePipeline` to generate the pipeline stages and the object `padv.pipeline.JenkinsOptions` to specify the pipeline generation options.
- 3 In your Jenkinsfile, update the file to use the:
 - Git branch, `credentialsId`, and `url` for your repository. For example:

```
git branch: 'testBranch',
credentialsId: 'jenkins-common-creds',
url: 'git://example.com/my-project.git'
```

- Path to the bin directory for your MATLAB installation. For example:

```

env.PATH = "C:\\Program Files\\MATLAB\\R2024b\\bin;${env.PATH}" // Windows
// env.PATH = "/usr/local/MATLAB/R2024b/bin:${env.PATH}" // Linux
// env.PATH = "/Applications/MATLAB_R2024b.app/bin:${env.PATH}" // macOS

withEnv(["PATH=C:\\Program Files\\MATLAB\\R2024b\\bin;${env.PATH}"]) { // Windows
// withEnv(["PATH=/usr/local/MATLAB/R2024b/bin:${env.PATH}"]) { // Linux
// withEnv(["PATH=/Applications/MATLAB_R2024b.app/bin:${env.PATH}"]) { // macOS

```

Now your Jenkinsfile file contains the Git repository information and path to the MATLAB installation for your CI setup.

```

//Scripted Pipeline
node {

    stage('Git Clone'){
        git branch: '#### Enter your branch ####',
        credentialsId: '#### Enter your credentials, if any ####',
        url: '#### Enter your repository URL ####'
    }

    // Requires MATLAB plugin
    stage('Pipeline Generation'){
        env.PATH = "C:\\Program Files\\MATLAB\\R2022a\\bin;${env.PATH}"

        /* Open the project and generate the pipeline using
        appropriate options */

        runMATLABCommand '''cp = openProject(pwd);
        padv.pipeline.generatePipeline(...
        padv.pipeline.JenkinsOptions(...
        PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask,...
        GeneratedJenkinsFileName = "simulink_pipeline",...
        GeneratedPipelineDirectory = fullfile("derived","pipeline"));'''
    }

    // pass necessary environment variables to generated pipeline
    withEnv(["PATH=C:\\Program Files\\MATLAB\\R2022a\\bin;${env.PATH}"]) {

        def rootDir = pwd()

        /* This file is generated automatically by
        padv.pipeline.generatePipeline with a default name
        of simulink_pipeline. Update this field if the
        name or location of the generated pipeline file is changed */

        load "${rootDir}/derived/pipeline/simulink_pipeline"
    }
}

```

Git Repository Information

Environment Path

Pipeline Generator

Environment Path
(for generated pipeline)

- 4 Add the Jenkinsfile to your project.
- 5 Push the changes to your project to source control.
- 6 Configure your Jenkins project to use the Jenkinsfile in source control.
 - a In the **Pipeline** section of the project configuration window, select Pipeline script from SCM from the **Definition** list.
 - b Select your source control system from the **SCM** list.
 - c Paste your repository URL into the **Repository URL** box.

For more information, see Plugin Configuration Guide (GitHub).

At this point, the template file is set up to generate a Jenkins pipeline, with stages for each task in your process, the next time that you trigger a build. Optionally, you can further customize the template file to change how the pipeline generator organizes and executes the pipeline. You can dry run your tasks, have separate stages for each task iteration in the process, and specify other options by using the `padv.pipeline.JenkinsOptions` object in the template.

Make Optional Customizations

Optionally, you can reconfigure the template file to customize how the pipeline generator organizes and executes the pipeline. To customize the pipeline generator options, you modify the property values of the `padv.pipeline.JenkinsOptions` object in the template.

For example, suppose that you want to:

- Dry run tasks to quickly validate task inputs and generate representative outputs without performing the full task operation.
- Perform license checkouts during the dry runs to make sure that your Jenkins agent has access to the required products.
- Have stages for each task iteration in the process.

To change how the template file generates the pipeline, you can modify the `padv.pipeline.JenkinsOptions` in the `runMATLABCommand` in the Pipeline Generation stage.

```
// Requires MATLAB plugin
stage('Pipeline Generation'){

    env.PATH = "C:\\Program Files\\MATLAB\\R2024b\\bin;${env.PATH}" // Windows
    // env.PATH = "/usr/local/MATLAB/R2024b/bin:${env.PATH}" // Linux
    // env.PATH = "/Applications/MATLAB_R2024b.app/bin:${env.PATH}" // macOS

    /* Open the project and generate the pipeline using
    appropriate options */

    runMATLABCommand '''cp = openProject(pwd);
    rpo = padv.pipeline.RunProcessOptions;
    rpo.DryRun = true;
    rpo.DryRunLicenseCheckout = true;
    padv.pipeline.generatePipeline(...
    padv.pipeline.JenkinsOptions(...
    RunprocessCommandOptions = rpo,...
    PipelineArchitecture = padv.pipeline.Architecture.SerialStages,...
    GeneratedJenkinsFileName = "simulink_pipeline",...
    GeneratedPipelineDirectory = fullfile("derived","pipeline"));'''
}
```

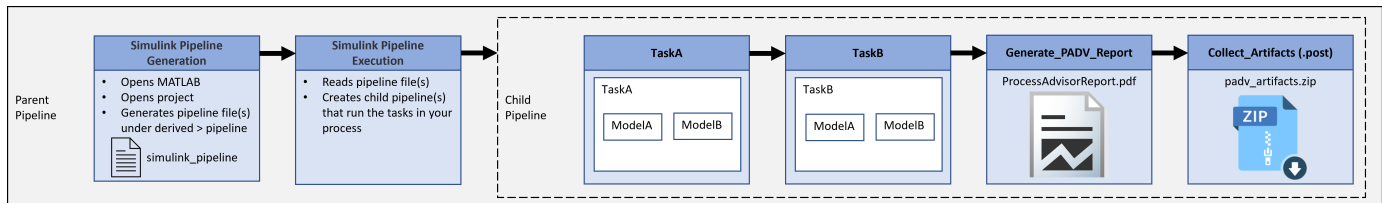
This example code creates a `padv.pipeline.RunProcessOptions` object, `rpo`, for customizing the behavior of the `runprocess` function in CI. In this case, specifying the `runprocess` arguments `DryRun` and `DryRunLicenseCheckout` as `true`. The code updates the `padv.pipeline.JenkinsOptions` object to use a different `PipelineArchitecture`, `SerialStages`, and use the `RunprocessCommandOptions` specified by `rpo`. For more information, see “How Pipeline Generation Works” on page 3-21.

If you modify other parts of the template file, make sure that your changes use valid Jenkins pipeline syntax. For more information, see the Jenkins documentation for Pipeline Syntax.

Generate Pipeline in Jenkins

The next time that you trigger a build, your generated pipeline contains two upstream stages in Jenkins:

- **Git_Clone** — Loads your Git repository information.
- **Pipeline Generation** — Automatically generates and loads a pipeline file called `simulink_pipeline` that defines downstream stages for your process. By default, the downstream stages include:
 - Stages for your process, organized by the `PipelineArchitecture` property specified in `padv.pipeline.JenkinsOptions`.
 - The `Generate_PADV_Report` stage, which generates a Process Advisor build report.
 - The stage, `Collect_Artifacts`, which collects build artifacts.



See Also

`padv.pipeline.generatePipeline` | `padv.pipeline.JenkinsOptions`

Related Examples

- “Approaches to Running Processes in CI” on page 3-2
- “How Pipeline Generation Works” on page 3-21
- “Tips for Setting Up CI Agents” on page 3-28

Integrate Process into Other CI Platforms

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team and run that process as a pipeline in CI. You can use any of the MATLAB-supported Continuous Integration (CI) platforms to run your automated pipeline of tasks. For more information on the supported platforms, see “Continuous Integration with MATLAB on CI Platforms”.

Before You Integrate

Your process model file defines the pipeline of tasks that the `runprocess` function runs in CI. If you do not already have a process model, the support package includes process model templates that you can use to get started. For more information, see “Automate and Run Tasks with Process Advisor” on page 1-2.

Before you try to run your process as part of an automated pipeline of tasks in CI, you need to connect your CI platform, remote repository, and project.

- 1 Create a remote repository for your project. Many CI platforms provide source-controlled remote repositories as part of their platform. But for other CI platforms you might need to host your remote repository on another platform. See the documentation for your chosen CI platform to identify how you want to set up your remote repository.
- 2 Set up a CI agent. Your CI agent machine is responsible for running MATLAB and communicating the results back to your chosen CI platform. Depending on the CI platform, you can set up the platform to run MATLAB on your own, self-hosted machine or in the cloud. Make sure that your CI agent can run MATLAB and that you install the support package and any other products required by your process. For information on licensing considerations, Docker containers, and virtual displays, see “Tips for Setting Up CI Agents” on page 3-28.
- 3 Connect your project, remote repository, and CI platform. On the **Project** tab, in the **Source Control** section, click **Remote** and specify the URL for your remote repository. For more information, see “Use Source Control with MATLAB Projects”.
- 4 Make sure that your process model file is available on the MATLAB path for your CI agent. As a best practice, keep your process model file in the project root folder and add the process model file to the project.

Run MATLAB in Batch Mode

To run your process in CI, you can use the `matlab` command with the `-batch` option in your CI system. You can use `matlab -batch` to run MATLAB code, including the `runprocess` function, noninteractively. For example, you can start MATLAB noninteractively, open the project, and run each of the tasks in the pipeline defined by the process model file (`processmodel.p` or `processmodel.m`) in the project.

```
matlab -batch "openProject(pwd);[~,exitCode] = runprocess();exit(exitCode);"
```

MATLAB terminates automatically with the exit code 0 if the specified code executes successfully without generating an error. Otherwise, MATLAB terminates with a nonzero exit code.

See Also

`matlab` | `runprocess`

Related Examples

- “Approaches to Running Processes in CI” on page 3-2
- “Best Practices for Effective Builds” on page 3-32
- “Tips for Setting Up CI Agents” on page 3-28

How Pipeline Generation Works

Typically, when you configure a CI pipeline, you need to manually create and update pipeline configuration files as you add, remove, and change the artifacts in your project. However, the example pipeline configuration files use a pipeline generator function, `padv.pipeline.generatePipeline`, that can generate the updated pipeline configuration files for you. After you do the initial setup for the pipeline generator, you do not need to manually update your pipeline configuration files. When you trigger your pipeline, the pipeline generator uses the digital thread to analyze the files in your project and uses your process model to generate pipeline configuration files for you.

Summary of Support

When you use the support package to integrate a model-based design (MBD) project into CI, there are three main approaches to creating and maintaining your pipeline configuration files:

- **Manual Authoring** — Each time you need to create or update your pipeline, you manually write, update, and check-in a pipeline configuration file that uses the `runprocess` function to run tasks. This approach allows you the most flexibility and ability to customize your pipeline, but requires that you regularly maintain the pipeline configuration file.
- **Manual Generation** — Each time you commit changes, you manually generate a pipeline configuration file using the `padv.pipeline.generatePipeline` function in your local MATLAB installation and then manually check the pipeline configuration file into your CI system. With this approach, you do not need to manually write the pipeline configuration file, but you do need to manually regenerate the pipeline for each submission.
- **Automatic Generation** — You perform a one-time setup of a parent pipeline configuration file that automatically calls the `padv.pipeline.generatePipeline` function and automatically generates an up-to-date, child pipeline configuration file that runs your process in CI. With this approach, you do not have to manually write or generate pipeline configuration files, but setting up a branching workflow can be complex.

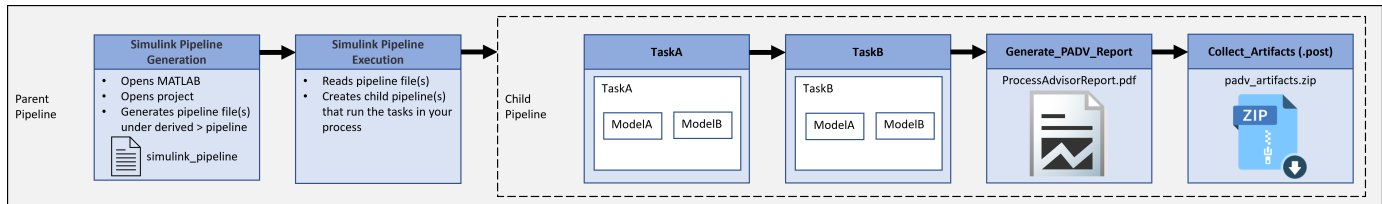
The following table lists which approaches the support package supports on each CI platform.

Approaches \ Platforms	GitHub	GitLab	Jenkins	Other MATLAB-Supported CI Platforms
Manual Authoring	✓	✓	✓	✓
Manual Generation	✓ (recommended)	✓	✓	
Automatic Generation		✓ (recommended)	✓ (recommended)	

For CI platforms, you typically define your CI pipeline by using a pipeline configuration file. For example, a YAML file on platforms like GitHub and GitLab or a `Jenkinsfile` on Jenkins.

Typically, when you configure a CI pipeline, you need to manually create and update your pipeline configuration files as you add, remove, and change the artifacts in your project. However, the support package has a pipeline generator function `padv.pipeline.generatePipeline` that you can use to generate the pipeline configuration files for GitHub, GitLab, and Jenkins.

For example, on a CI platform like GitLab, the pipeline generator can automatically generate the pipeline configuration files that you would need to create a pipeline that runs each job in your process, generate a report, and collect the artifacts from the pipeline.



Generated Pipelines

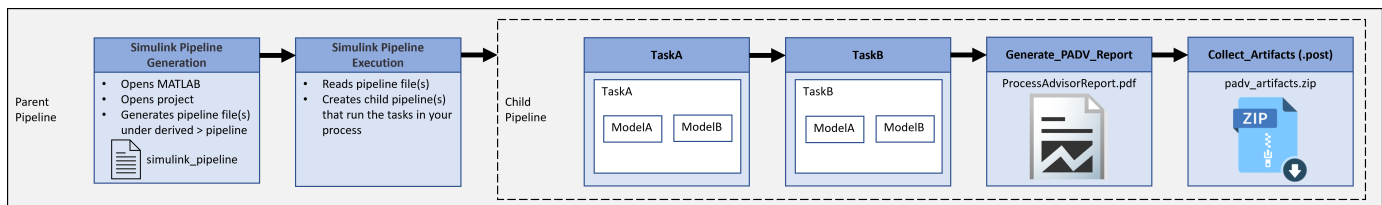
After you perform the initial setup and trigger your pipeline, the pipeline generator generates a parent pipeline and a child pipeline.

The parent pipeline contains two stages:

- **Simulink Pipeline Generation** — This stage analyzes your project and process model to automatically generate the pipeline configuration files to run your process in CI. If you want to view the generated pipeline configuration files, the pipeline generator stores the files under the `derived > pipeline` folder in the project.
- **Simulink Pipeline Execution** — This stage creates and executes a child pipeline that runs the tasks in your process, generates a build report, and collects the job artifacts.

By default, the child pipeline contains:

- One stage for each task in your process model.
- One stage that generates a build report, `ProcessAdvisorReport.pdf`.
- One stage that collects the job artifacts and compresses the artifacts into a zip file, `padv_artifacts.zip`.



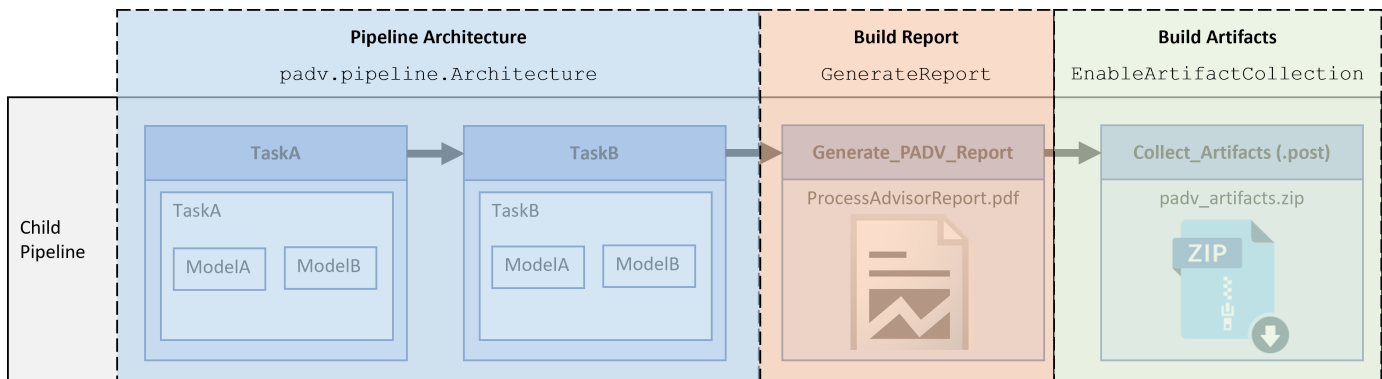
Optional Pipeline Customization

You can run the pipeline generator using the default options or you can edit the example pipeline configuration file to customize how the pipeline generator creates and executes pipelines in CI.

The call to the pipeline generator function (`padv.pipeline.generatePipeline`) is in the example pipeline configuration file. The function `padv.pipeline.generatePipeline` requires you to specify a CI options object as an input.

The CI options object allows you to specify several properties of the generated CI pipeline, including:

- the pipeline architecture
- whether the pipeline generates a build report
- if and when the pipeline collects artifacts from the build



Pipeline Architecture

The pipeline architecture defines the number of stages and the grouping of tasks in the child pipeline. You can specify the pipeline architecture by using a `padv.pipeline.Architecture` object.

By default, the example pipeline configuration files specify the pipeline architecture as `SerialStagesGroupPerTask`, which creates one stage for each task in the process model. For example, one stage for `TaskA` and one stage for `TaskB`.

The available pipeline architectures are:

- `SingleStage` — A single stage, **Runprocess**, that runs all the tasks in the process.
- `SerialStages` — One stage for each task iteration in the process.
- `SerialStagesGroupPerTask` — One stage for each task in the process.
- `IndependentModelPipelines` — Parallel, downstream pipelines for each model. Each pipeline independently runs the tasks associated with the model. For information how parallel pipeline architecture work and process considerations, see “Parallel Pipeline Architectures” on page 3-24.

Build Report

By default, the pipeline generator creates a stage, **Generate_PADV_Report**, that generates a build report for your pipeline. The build report is a PDF file `ProcessAdvisorReport.pdf`.

If you do not want to generate a report, you can specify the `GenerateReport` argument as `false`. For example, in a GitLab pipeline configuration file:

```
padv.pipeline.GitLabOptions(GenerateReport = false)
```

Build Artifacts

By default, the pipeline generator creates a stage, **Collect_Artifacts**, that collects and compresses the build artifacts from your pipeline. The ZIP file attached to the **Collect_Artifacts** stage is called `padv_artifacts.zip`. You can download these artifacts to locally reproduce issues seen in CI. For more information, see “Locally Reproduce Issues Found in CI” on page 1-20.

You can specify if and when you want the pipeline to collect artifacts by specifying the argument `EnableArtifactCollection`:

- "never", 0, or false — Never collect artifacts
- "on_success" — Only collect artifacts when the pipeline succeeds
- "on_failure" — Only collect artifacts when the pipeline fails
- "always", 1, or true — Always collect artifacts

For example, in a GitLab pipeline configuration file:

```
padv.pipeline.GitLabOptions(EnableArtifactCollection="on_failure")
```

Parallel Pipeline Architectures

Starting in R2023b Update 5, the pipeline generator supports a round-trip, parallel CI workflow that automatically merges the task statuses and project analysis performed in parallel. By default, Process Advisor and the build system store task statuses and project analysis in an artifact database file, `artifacts.dmr`. If you use a parallel pipeline architecture like `IndependentModelPipelines`, the pipeline generator needs to merge artifact database files from across different parallel jobs. Depending on your process model, the pipeline generator can automatically add these stages to the generated pipeline:

- **Create_Base_Artifact_Database** — Before running your parallel jobs, the pipeline generator creates a common ancestor artifact database file, `base.dmr`, that the pipeline generator can use when merging the task statuses and project analysis performed in the parallel. This stage uses the utility function `padv.util.saveArtifactDatabase` to save a copy of the artifact database file.
- **Merge_Artifact_Databases** — After running your parallel jobs, the pipeline generator merges the artifact database files created by each parallel branch with the common ancestor artifact database file `base.dmr`. This stage uses the utility function `padv.util.mergeArtifactDatabases` to merge the artifact database files into a single `artifacts.dmr` file that contains the information from the parallel branches.

If your process model includes code generation and code analysis, the pipeline generator can automatically merge the artifact database files as part of your top model code generation stage. For more information, see “Considerations for Parallel Code Generation”.

When you download your CI artifacts onto your machine, this merged `artifacts.dmr` file allows you to see up-to-date task statuses locally in Process Advisor. The **Collect_Artifacts** stage automatically includes the `artifacts.dmr` file inside the `derived` folder in the `artifacts.zip` file.

Considerations for Parallel Code Generation

Starting in R2023b Update 5, if you want to use a parallel pipeline architecture and your process contains code generation and code analysis tasks, you need to either use the example parallel process model or update your existing process model. These updates allow the tasks in your pipeline to properly handle shared utilities and code generated across parallel jobs.

Example Parallel Process Model

To see the example parallel process model, you can either:

- Open the Process Advisor example project for parallel pipelines:

```
processAdvisorParallelExampleStart
```

- Create a parallel process model using the parallel template:

```
createprocess(Template = "parallel")
```

Update Existing Process Model

To update your existing process model for a round-trip parallel CI workflow, you need to:

- Have a task that generates code for your reference models. The task must specify the property `GenerateExternalCodeCache` as `true` and specify an `ExternalCodeCacheDirectory`. The external code cache allows your team to generate code in parallel while maintaining up-to-date task status information. For example:

```
% Generate Code for Reference Models
codegenTask = pm.addTask(padv.builtin.task.GenerateCode("IterationQuery", ...
    padv.builtin.query.FindRefModels));
codegenTask.UpdateThisModelReferenceTarget = 'IfOutOfDate';
codegenTask.TreatAsRefModel = true;
codegenTask.Title = "Reference Model Code Generation";
codegenTask.GenerateExternalCodeCache = true;
codegenTask.ExternalCodeCacheDirectory = fullfile( ...
    '$DEFAULTOUTPUTDIR$', '$ITERATIONARTIFACT$', 'external_code_cache');
```

- Have a task that generates code for your top models. The task must iterate over the project file, specify the property `GenerateExternalCodeCache` as `true`, and specify an `ExternalCodeCacheDirectory`. The external code cache allows your team to generate code in parallel while maintaining up-to-date task status information. For example:

```
% Generate Code for Top Models (at the project-level)
codegenTopTask = pm.addTask(padv.builtin.task.GenerateCode("IterationQuery", ...
    padv.builtin.query.FindProjectFile,"InputQueries",...
    {padv.builtin.query.FindTopModels,...
    padv.builtin.query.GetOutputsOfDependentTask(...
    "padv.builtin.task.GenerateCode")},...
    "Name", "Top Model Code Generation"));
codegenTopTask.UpdateThisModelReferenceTarget = 'IfOutOfDate';
codegenTopTask.TreatAsRefModel = false;
codegenTopTask.Title = "Top Model Code Generation";
codegenTopTask.TrackAllGeneratedCode = true;
```

- Split code analysis tasks into two tasks. One task for reference models and one task for top models. The task for top models must iterate over the project file. The built-in code analysis tasks, like `padv.builtin.task.RunCodeInspection`, are able to unpack the code generation target from the external code cache by using the utility function `padv.util.unpackExternalCodeCache`.

```
% Inspect Generated Code for Reference Models
slciTask = pm.addTask(padv.builtin.task.RunCodeInspection("IterationQuery", ...
    padv.builtin.query.FindRefModels));
slciTask.ReportFolder = fullfile(defaultResultPath,'code_inspection');
slciTask.Title = "Ref Model Code Inspection";
```

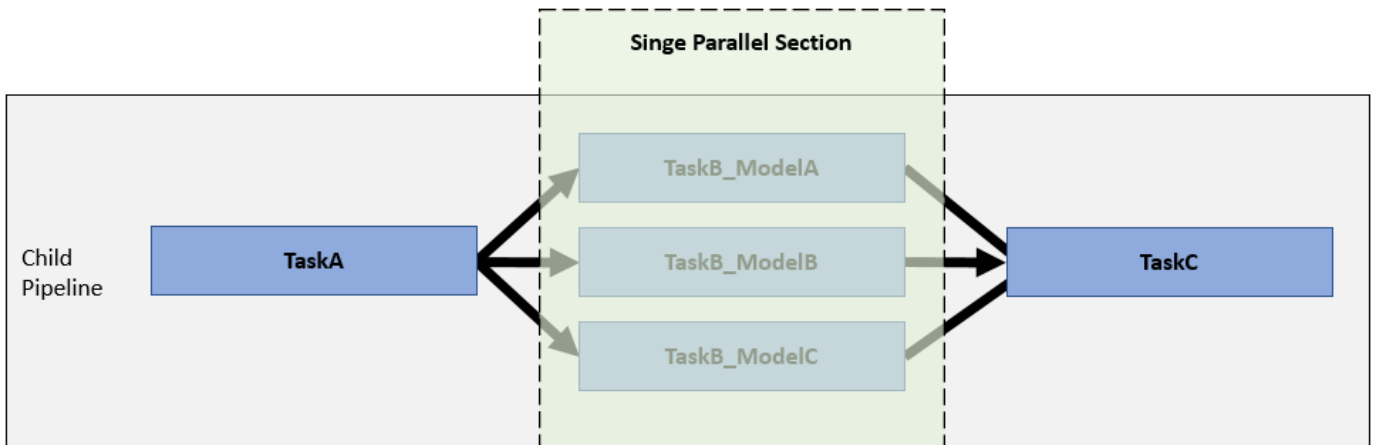
```
% Inspect Generated Code for Top Models (at the project-level)
slciTopTask = pm.addTask(padv.builtin.task.RunCodeInspection("IterationQuery", ...
    padv.builtin.query.FindProjectFile,"InputQueries",...
    {padv.builtin.query.GetOutputsOfDependentTask("Top Model Code Generation"),...
    padv.builtin.query.FindTopModels},"Name","Top Model Code Inspection"));
slciTopTask.Title = "Top Model Code Inspection";
slciTopTask.ReportFolder = fullfile('$DEFAULTOUTPUTDIR$', 'code_inspection',...
```

```

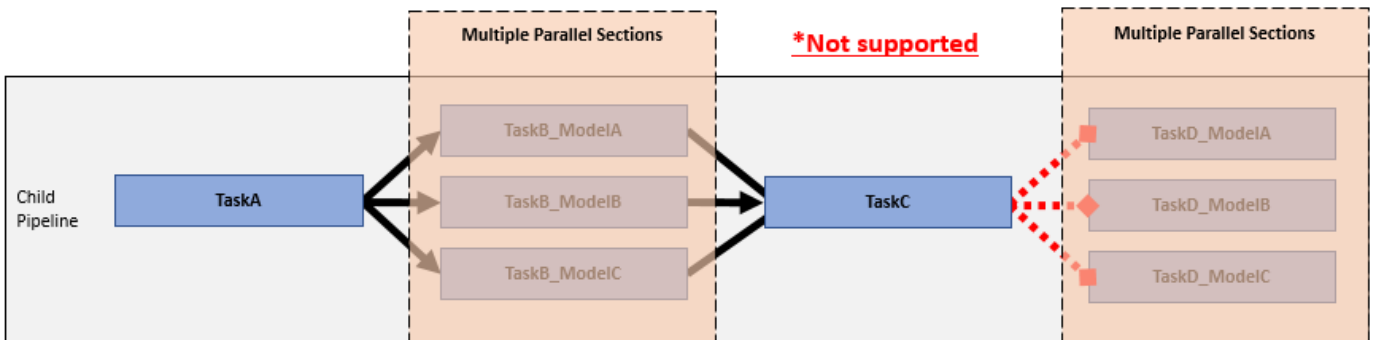
'$INPUTARTIFACT$');
slciTopTask.OutputDirectory = string(fullfile('$DEFAULTOUTPUTDIR$', 'code_inspection'))
    
```

- Update the dependsOn and runsAfter relationships in your process model to specify the relationships for these tasks.

Note There are limitations to the task relationships that the pipeline generator can support. The pipeline generator requires your process model to only generate one parallel section. If tasks, like model tasks, run in parallel, you must define your task relationships so that all subsequent tasks iterate over the project file. The pipeline generator only supports a single shift from parallel to serial execution per CI build because the pipeline generator only merges the artifact database files once.



The pipeline generator does not support, for example, a process model that alternates between tasks that execute in parallel, then in serial, then parallel again.



The example parallel process model uses top model code generation and code analysis tasks that iterate over the project file to avoid creating multiple parallel sections.

See Also

padv.pipeline.generatePipeline | padv.pipeline.GitHubOptions |
 padv.pipeline.GitLabOptions | padv.pipeline.JenkinsOptions

Related Examples

- “Approaches to Running Processes in CI” on page 3-2
- “Best Practices for Effective Builds” on page 3-32
- “Tips for Setting Up CI Agents” on page 3-28

Tips for Setting Up CI Agents

A CI agent is a machine that is responsible for running MATLAB and communicating the results back to your chosen CI platform. Depending on the CI platform, you might set up the platform to run MATLAB on your own, self-hosted machine or in the cloud. Use the following suggestions to help set up your CI agent.

Product Installation

To use the support package in CI, you need to install at least these products on your CI agent:

- MATLAB
- Simulink
- Simulink Check
- CI/CD Automation for Simulink Check
- Any other products required by your process

You can programmatically install products by using the MATLAB Package Manager (MPM). For more information, see <https://github.com/mathworks-ref-arch/matlab-dockerfile/blob/main/MPM.md>.

Note License Considerations for CI: If you plan to perform CI on many hosts or in the cloud, contact MathWorks® (continuous-integration@mathworks.com) for help. Transformational products such as MathWorks coder and compiler products might require client access licenses (CAL).

Dry Run Your Process

Before you try to run your process on your CI agent, you can dry run your process. The dry run can validate your task inputs, generate representative task outputs, and make sure that you have the required licenses available on your CI agent.

To perform a dry run, you can use the `DryRun` argument of the `runprocess` function. For example:

```
runprocess(DryRun = true)
```

To automatically check out the licenses associated with the tasks, you can specify the `DryRunLicenseCheckout` argument as `true`:

```
runprocess(DryRun = true, DryRunLicenseCheckout = true)
```

Dry runs can be helpful for quickly testing out your CI pipeline and making sure that your required products and licenses are available, locally and on your CI agents. The built-in tasks have a `dryRun` method that generates representative task outputs for that task. You can define your own custom dry run functionality by overriding the `dryRun` method for class-based tasks or specifying the task property `DryRunAction` for function-based tasks.

For more information on dry runs, see “Dry Run Tasks to Test Process Model” on page 2-66.

Set Up Virtual Display Machines Without Displays

Some MATLAB code, including some built-in tasks, can only run successfully if a display is available for your machine. When there is no display available, MATLAB returns an error.

A machine might not have a display available if either:

- You start MATLAB using the `-nodisplay` option.
- The machine does not have a display configured and the `DISPLAY` environment variable is not set. For example:
 - some CI runners
 - some containers, including Docker containers by default
 - machines that you SSH into without X11 forwarding

If MATLAB returns an error related to your display, try the following workaround.

You can set up a virtual display on the machine to simulate a display environment. The virtual display allows you to run MATLAB code that requires a display, without having to connect your machine to a physical display.

- 1 Choose a server. There are several common servers that you can install and use to host your virtual display, including:
 - Xvfb — <https://manpages.ubuntu.com/manpages/trusty/man1/xvfb-run.1.html>
 - VNC server — <https://help.ubuntu.com/community/VNC/Servers>
- 2 Install the server on the machine. For example, to install Xvfb on a Linux® machine:

```
sudo apt-get install xvfb
```

Alternatively, for a containerized environment, you can instruct your container image to install and use the server as the display. For example, to install and use Xvfb for a Docker container, your Dockerfile can include:

```
RUN apt-get install --no-install-recommends --yes xvfb
RUN export DISPLAY=:Xvfb -displayfd 1 &` && \
```

Tip To access an example Dockerfile that uses Xvfb, enter the following command in MATLAB:

```
cd(fullfile(matlabshared.supportpkg.getSupportPackageRoot, ...
"toolbox", "padv", "samples"))
```

- 3 Run MATLAB in the server environment.

For example, with Xvfb on a Linux machine, you can use the `xvfb-run` command to run your MATLAB code with a virtual display. For example:

```
xvfb-run matlab -batch "openProject(projectPath);runprocess;"
```

For information, see <https://manpages.ubuntu.com/manpages/trusty/man1/xvfb-run.1.html>.

Note Depending on which server you choose, you might need to manually start the server and set the `DISPLAY` environment variable on your machine to use your virtual display. The `DISPLAY` environment variable cannot be left empty.

Since most CI runners and containers do not have a display available, you should set up a virtual display server before you include the following built-in tasks in your process model:

- `padv.builtin.task.GenerateSDDReport`
- `padv.builtin.task.GenerateSimulinkWebView`
- `padv.builtin.task.GenerateModelComparison`

Create Docker Container for Support Package

A container is an isolated unit of software that contains everything required to run a specific application. You can use a container as a scalable and reproducible way to deploy and test your process.

You can follow these steps to create a Docker image that includes MATLAB, other MathWorks products, and the CI/CD Automation for Simulink Check support package. The example Dockerfile installs the support package and other products by using the MATLAB Package Manager (MPM). Since certain MATLAB code requires a display to run successfully, the example Dockerfile uses Xvfb to set up a virtual display for the container.

The MATLAB Docker image is a Linux executable, but can run on any host operating system that Docker supports. For general information about MATLAB container images, see <https://github.com/mathworks-ref-arch/matlab-dockerfile>.

- 1 Install Docker on your machine. For information, see <https://docs.docker.com/get-docker/>.
- 2 To access the example Dockerfile for Process Advisor, open MATLAB and enter:

```
open(fullfile(matlabshared.supportpkg.getSupportPackageRoot, ...  
"toolbox", "padv", "samples", "Dockerfile"))
```

- 3 Save a copy of the file, `Dockerfile` (no file extension), in a directory that your Docker daemon can access.
- 4 Build a Docker image by using the `docker build` command. You can use the build-time variables to specify the MATLAB release, MathWorks products, installation location, network license, and name for your container image. For example:

```
docker build --build-arg MATLAB_RELEASE=2023b  
--build-arg PRODUCTS="MATLAB Simulink Simulink_Check CI/CD_Automation_for_Simulink_Check"  
--build-arg MATLAB_INSTALL_LOCATION="/opt/matlab/R2023b"  
--build-arg LICENSE_SERVER=port@hostname  
-t my_matlab_image_name .
```

This example code only installs the products required by the support package. If you want to be able to run all of the built-in tasks, see the example Dockerfile for a list of the other products to add to the `PRODUCTS` list.

Use the build-arg `LICENSE_SERVER` to specify the port and hostname for your network license manager.

Alternatively, you can place your `network.lic` file in the same folder as the example Dockerfile, uncomment the line `COPY network.lic ${MATLAB_INSTALL_LOCATION}/licenses` in the

example Dockerfile, and run the `docker build` command without the `LICENSE_SERVER` build-arg.

```
docker build --build-arg MATLAB_RELEASE=2023b
--build-arg PRODUCTS="MATLAB Simulink Simulink_Check CI/CD_Automation_for_Simulink_Check"
--build-arg MATLAB_INSTALL_LOCATION="/opt/matlab/R2023b"
-t my_matlab_image_name .
```

For more information, see <https://docs.docker.com/reference/cli/docker/image/build/> and <https://github.com/mathworks-ref-arch/matlab-dockerfile>.

Note The example Dockerfile assumes that you are using the network license manager to license and run MATLAB. If you run MATLAB using a different licensing approach, contact MathWorks (continuous-integration@mathworks.com) for help.

- 5 Create and run a container from the generated image by using the `docker run` command.

```
docker run --init --rm my_matlab_image_name -batch "ver"
```

For information, see <https://docs.docker.com/reference/cli/docker/container/run/>.

See Also

`runprocess`

Related Examples

- “Approaches to Running Processes in CI” on page 3-2
- “How Pipeline Generation Works” on page 3-21

Best Practices for Effective Builds

With the CI/CD Automation for Simulink Check support package, you can define a development and verification process for your team by using a process model. When you deploy your process model to your team, consider the following best practices for scheduling builds and caching artifacts.

Use Incremental Builds for Regular Submissions

For builds that you perform on a daily or more frequent basis, use incremental builds. Incremental builds are faster and more efficient, but incremental builds skip tasks that the build system considers up to date.

By default, the function `runprocess` performs an incremental build:

```
runprocess()
```

If you use a pull request workflow, incremental builds are helpful for efficiently prequalifying changes before merging with the main repository.

Run Full Builds for Qualifying Software

Outside of the normal build schedule, you should run a full (non-incremental) build at least one time per week and anytime you are qualifying software for a release. When you run a full build, the build system force runs each of the tasks in the pipeline. The full build makes sure that each task in the pipeline executes and that the output artifacts reflect the latest changes.

To run a full build, use the function `runprocess` with the argument `Force` specified as `True`:

```
runprocess(Force=true)
```

The `Force` argument forces tasks in the pipeline to execute, even if the tasks already have up to date results.

For more information, see “Specify Settings for Process Advisor and Build System” on page 1-16 and `runprocess`.

Cache Models and Other Artifacts Used During Build

If you select the setting **Enable model caching**, the build system can cache your models and several other artifacts. The cache allows the build system to avoid reloading the same artifacts multiple times within a build.

The artifacts that the build system can cache include:

- Simulink models
- Simulink libraries, subsystem references, and data dictionaries
- Test files, results, and harnesses (internally saved and externally saved) from Simulink Test
- Requirements files and requirement sets from Requirements Toolbox™
- System Composer architecture models

You can control the size of the cache by using the `padv.ProjectSettings` properties `MaxNumModelsInCache` and `MaxNumTestResultsInCache`. The built-in tasks use the utility

function `padv.util.closeModelsLoadedByTask` to close models loaded by the task. For more information, see `padv.ProjectSettings` and `padv.util.closeModelsLoadedByTask`.

If you have custom tasks, you can improve the efficiency of model loading in your builds by closing the models loaded by a task by using the function `padv.util.closeModelsLoadedByTask` inside your custom tasks.

For example:

```
classdef MyCustomTask < padv.Task
    methods
        function obj = MyCustomTask(options)
            arguments
                % unique identifier for task
                options.Name = "MyCustomTask";
                % artifacts the task iterates over
                options.IterationQuery = "padv.builtin.query.FindModels";
                % input artifacts for the task
                options.InputQueries = "padv.builtin.query.GetIterationArtifact";
                % where the task outputs artifacts
                options.OutputDirectory = fullfile(...
                    '$DEFAULTOUTPUTDIR$', 'my_custom_task_results');
            end
            % Calling constructor of superclass padv.Task
            obj@padv.Task(options.Name,...
                IterationQuery=options.IterationQuery,...
                InputQueries=options.InputQueries);
            obj.OutputDirectory = options.OutputDirectory;
        end
        function taskResult = run(obj,input)
            % Before the task loads models, save a list of the models that are already loaded.
            loadedModels = get_param(Simulink.allBlockDiagrams(), 'Name');

            % identify model name
            % "input" is a cell array of input artifacts
            % First input query gets iteration artifact (a model)
            model = input{1}; % get padv.Artifact from first input query
            modelName = padv.util.getModelName(model);

            % Example task that loads model and displays information
            load_system(modelName);
            disp(modelName);
            disp('Data Dictionaries:')
            disp(Simulink.data.dictionary.getOpenDictionaryPaths)

            % specify results from task using padv.TaskResult
            taskResult = padv.TaskResult;
            taskResult.Status = padv.TaskStatus.Pass;
            % taskResult.Status = padv.TaskStatus.Fail;
            % taskResult.Status = padv.TaskStatus.Error;

            % Close models that were loaded by this task.
            padv.util.closeModelsLoadedByTask(...
                PreviouslyLoadedModels=loadedModels)
        end
    end
end
end
```

See Also

`padv.ProjectSettings` | `padv.UserSettings` | `padv.util.closeModelsLoadedByTask` | `runprocess`

Related Examples

- “Best Practices for Process Model Authoring” on page 2-56
- “Manage Multiple Build and Verification Workflows Using Processes” on page 2-49
- “Specify Settings for Process Advisor and Build System” on page 1-16

Version History

- “September 2024” on page 4-2
- “July 2024” on page 4-5
- “June 2024” on page 4-7
- “May 2024” on page 4-10
- “April 2024” on page 4-13
- “March 2024” on page 4-14
- “February 2024” on page 4-20
- “December 2023” on page 4-23
- “November 2023” on page 4-25
- “October 2023” on page 4-27
- “September 2023” on page 4-29
- “August 2023” on page 4-31
- “July 2023” on page 4-32
- “June 2023” on page 4-33
- “April 2023” on page 4-36
- “March 2023” on page 4-39
- “February 2023” on page 4-40
- “December 2022” on page 4-41
- “November 2022” on page 4-42
- “October 2022” on page 4-43
- “September 2022” on page 4-44
- “August 2022” on page 4-45

▲ September 2024

Supported releases:

- R2024a
- R2023b
- R2023a¹

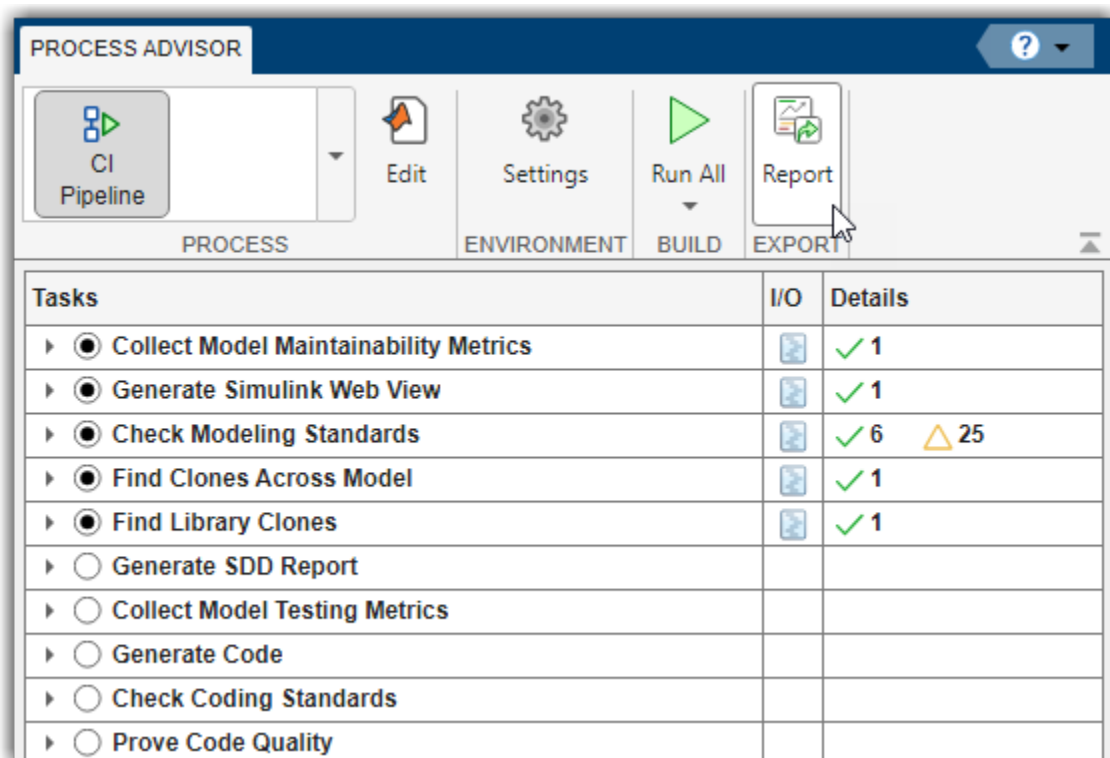
Documentation

The documentation for the CI/CD Automation for Simulink Check support package is now included in the online documentation for Simulink Check. See Continuous Integration.

Updates will be announced in the Simulink Check release notes and on the File Exchange page for the support package. The September release is the last planned release for the User's Guide and Reference Book PDFs that ship with the support package. To access the PDF Documentation for Simulink Check, see PDF Documentation for Simulink Check .

Features

- You can now export a report directly from Process Advisor by using the report button in the toolstrip.



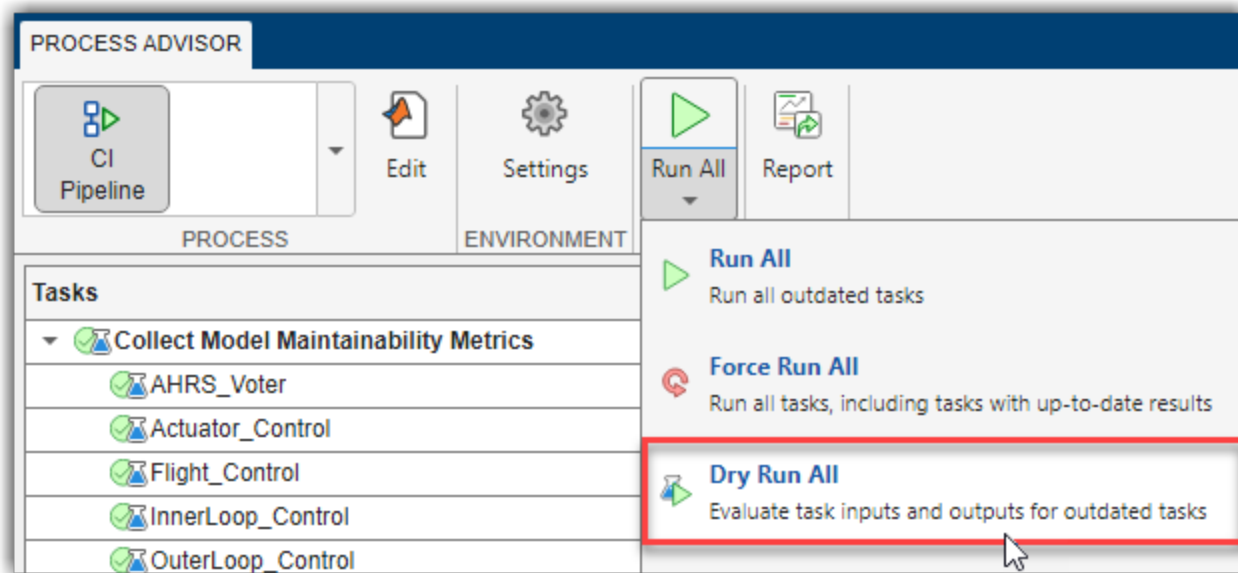
¹ The July release was the last planned release for R2022b.

- You can now dry run tasks directly from Process Advisor. Dry runs can help you test your process model by validating task inputs and generating representative task outputs without actually running the tasks.

In Process Advisor, you can:

- Dry run each task in the process by clicking **Run All > Dry Run All** in the toolstrip.
- Dry run a specific task by pointing to the task and clicking **... > Dry Run Task**

For more information, see “Dry Run Tasks to Test Process Model” on page 2-66.



- You can now control which processes are in your process model. Previously, the process model automatically always added a default CIPipeline process to your process model and you could not rename or remove that process. Now, if you add a process to your process model, the process model no longer automatically creates the CIPipeline process. For more information, see “Manage Multiple Build and Verification Workflows Using Processes” on page 2-49.
- You can use the MATLAB Unit Test framework to execute your test cases by specifying the new task property UseMATLABUnitTest for the built-in tasks RunTestsPerModel and RunTestsPerTestCase.

If you use the pipeline generator, `padv.pipeline.generatePipeline`, and your pipeline generator options specify the `GenerateJUnitForProcess` property as `true (1)`, the task uses the MATLAB unit test XML plugin to help you produce CI-compatible, JUnit-style XML report artifacts for your pipeline.

- The built-in tasks `MergeTestResults`, `RunTestsPerModel`, and `RunTestsPerTestCase` now generate more detailed JUnit-style XML reports so that you can view additional task result information in CI.
- You can choose whether the built-in task `DetectDesignErrors` generates a report by specifying the new task property `GenerateReport` as `true (1)` or `false (0)`. For more information, see `padv.builtin.task.DetectDesignErrors`.

Compatibility Considerations

- For the built-in tasks, the legacy options structures `RunOptions`, `ReportOptions`, `CoverageReportOptions`, `TestReportOptions`, and `ResultSavingOptions` have been removed. The legacy options structures were replaced by the built-in task properties in the April 2023 release. To reconfigure a built-in task, you must use the current task properties.

For example:

Functionality	Use This Instead
<code>maTask.RunOptions.ReportPath</code>	<code>maTask.ReportPath</code>

- Starting in R2024b, the built-in queries `FindRequirements` and `FindRequirementsForModel` will return artifacts of type `"mwreq_file"`. If you have queries that use the artifact types `"sl_req"` or `"sl_req_file"`, you need to update your code.
- The following `padv.Task` properties are now hidden:
 - `OutputQueries`
 - `IncludeMatlabWarningsInResults`
 - `Products`
 - `AllLicenseRequired`
 - `ConfigurationFileExtensions`
 - `EditConfigurationAction`

▲ July 2024

Supported releases:

- R2024b
- R2024a
- R2023b
- R2023a
- R2022b Update 1 (and later updates)²

Features

- You can now dry-run tasks to quickly validate your task inputs and generate representative task outputs without actually running the task action. To perform a dry-run, you can use the `DryRun` argument of the `runprocess` function. For example:

```
runprocess(DryRun = true)
```

To automatically check out the licenses associated with the tasks, you can specify:

```
runprocess(DryRun = true, DryRunLicenseCheckout = true)
```

Dry-runs can be helpful for quickly testing out your CI pipeline and making sure that your required products and licenses are available, locally and on your CI agents. The built-in tasks now have a `dryRun` method that generates representative task outputs for each task. You can define your own custom dry-run functionality by overriding the `dryRun` method for class-based tasks or specifying the task property `DryRunAction` for function-based tasks.

Compatibility Considerations

Previously, in your pipeline generator options object, you specified `runprocess` arguments by using these properties:

- `ForceRunAllTasks`
- `ExitInBatchMode`
- `RerunFailedTasks`
- `RerunErroredTasks`
- `GenerateJUnitForProcess`

These properties will be removed from the pipeline generator options objects in a future release. Use the new property `RunprocessCommandOptions` instead.

Object	Previous Properties	New Property
<code>padv.pipeline.GitHubOptions</code>	<ul style="list-style-type: none"> • <code>ForceRunAllTasks</code> • <code>ExitInBatchMode</code> • <code>RerunFailedTasks</code> • <code>RerunErroredTasks</code> 	<code>RunprocessCommandOptions</code>
<code>padv.pipeline.GitLabOptions</code>		

² The July release is the last planned release for R2022b.

Object	Previous Properties	New Property
<code>padv.pipeline.JenkinsOptions</code>	<ul style="list-style-type: none"> • <code>GenerateJUnitForProcess</code> 	

Instead of specifying `runprocess` arguments directly in your pipeline generator options object:

- 1 Create a `padv.pipeline.RunProcessOptions` object.
- 2 Set the properties of the object.
- 3 Use the object to specify the property `RunprocessCommandOptions` for your pipeline generator options object.

This example shows how to create a GitHub pipeline generator options object that specifies certain `runprocess` arguments using the recommended functionality.

Functionality	Use This Instead
<pre>padv.pipeline.GitHubOptions(... ForceRunAllTasks = false,... ExitInBatchMode = false,... RerunFailedTasks = false,... RerunErroredTasks = false,... GenerateJUnitForProcess = false);</pre>	<pre>op = padv.pipeline.RunProcessOptions; op.Force = false; op.ExitInBatchMode = false; op.RerunFailedTasks = false; op.RerunErroredTasks = false; op.GenerateJUnitForProcess = false; padv.pipeline.GitHubOptions(... RunprocessCommandOptions = op)</pre>

▲ June 2024

Supported releases:

- R2024a
- R2023b
- R2023a
- R2022b Update 1 (and later updates)

Features

- You can collect model design and testing metrics for the units and components in your project by using the new built-in task `padv.builtin.task.CollectMetrics`. These metrics correspond to the metrics in the Model Maintainability Dashboard and Model Testing Dashboard.

You can add tasks for collecting different metrics by using `addTask` and configuring the tasks inside your process model. By default, the `CollectMetrics` collects metrics for the Model Maintainability Dashboard, but you can reconfigure the task to collect model testing and code testing metrics by changing the iteration query and specifying the Dashboard property.

```

%% Collect Model Maintainability Metrics
mmMetricTask = pm.addTask(padv.builtin.task.CollectMetrics());

%% Collect Model Testing Metrics
mtMetricTask = pm.addTask(padv.builtin.task.CollectMetrics(...
    Name="ModelTestingMetrics",...
    IterationQuery=padv.builtin.query.FindUnits));
mtMetricTask.Title = "Collect Model Testing Metrics";
mtMetricTask.Dashboard = "ModelUnitTesting";
mtMetricTask.ReportName = "$ITERATIONARTIFACT$_ModelTesting";

%% Collect SIL Code Testing Metrics
stMetricTask = pm.addTask(padv.builtin.task.CollectMetrics(...
    Name="SILTestingMetrics",...
    IterationQuery=padv.builtin.query.FindUnits));
stMetricTask.Title = "Collect SIL Code Testing Metrics";
stMetricTask.Dashboard = "ModelUnitSILTesting";
stMetricTask.ReportName = "$ITERATIONARTIFACT$_SILTesting";

%% Collect PIL Code Testing Metrics
ptMetricTask = pm.addTask(padv.builtin.task.CollectMetrics(...
    Name="PILTestingMetrics",...
    IterationQuery=padv.builtin.query.FindUnits));
ptMetricTask.Title = "Collect PIL Code Testing Metrics";
ptMetricTask.Dashboard = "ModelUnitPILTesting";
ptMetricTask.ReportName = "$ITERATIONARTIFACT$_PILTesting";

```

When you point to one of the tasks in Process Advisor, you have the option to launch the associated dashboard (Model Design Dashboard or Model Testing Dashboard).

For more information, see `padv.builtin.task.CollectMetrics`, `padv.builtin.query.FindDesignModels`, and `padv.builtin.query.FindUnits`.

- To find models that are associated with test cases that use a specific test case tag, use the new Tags argument for the built-in query `padv.builtin.query.FindModelsWithTestCases`.
- Get the absolute path to an artifact by using the new object function `getAbsolutePath` for `padv.util.ArtifactAddress`.
- When you create a new process model with the `createprocess` function, you can now automatically set up the default process model template to groups model verification tasks and code verification tasks into separate subprocesses by specifying the Subprocess as true:

```
createprocess(Subprocess = true)
```

When you open Process Advisor, the **Tasks** column shows the tasks grouped into **Model Verification** and **Code Verification**.

Tasks	I/O	Details
<ul style="list-style-type: none"> ▼ ○ Model Verification ▶ ○ Collect Model Maintainability Metrics ▶ ○ Generate Simulink Web View ▶ ○ Check Modeling Standards ▶ ○ Generate SDD Report ▶ ○ Run Tests ▶ ○ Merge Test Results ▶ ○ Collect Model Testing Metrics 		
<ul style="list-style-type: none"> ▼ ○ Code Verification ▶ ○ Generate Code ▶ ○ Check Coding Standards ▶ ○ Prove Code Quality 		

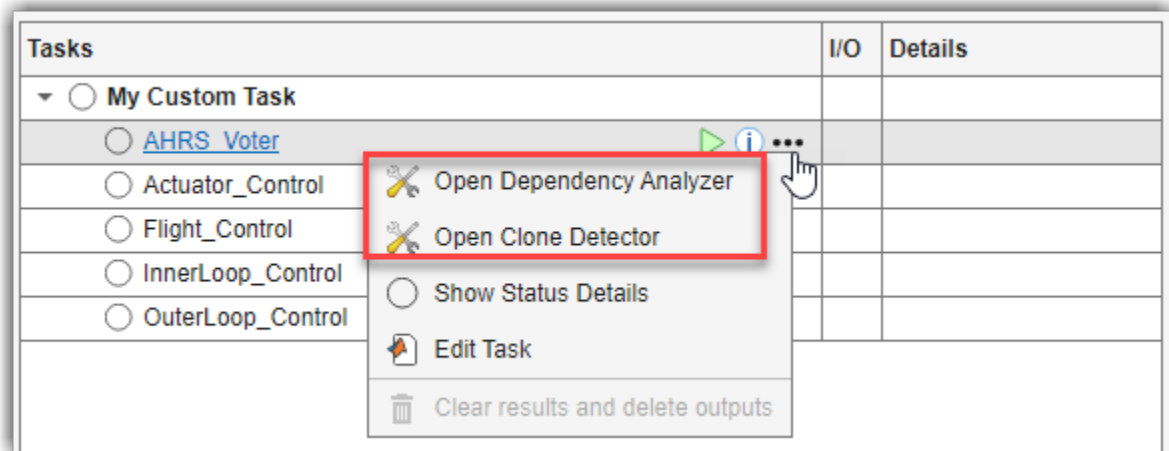
Additionally, if you want to create an instance of the Process Advisor example project that uses **Model Verification** and **Code Verification** subprocesses, you can use the Subprocess argument. For example:

```
processAdvisorExampleStart(Subprocess = true)
```

- You now have the option to open multiple tools from the options menu (...) of a task. To associate multiple tools with a task, specify the task property `LaunchToolAction` as a cell array of function handles and `LaunchToolText` as a string array. For each tool action that you specify in `LaunchToolAction`, you must have corresponding text specified in `LaunchToolText`. For example, to create a custom task that has options for opening the Dependency Analyzer app and the Clone Detector app:

```
t = addTask(pm, 'MyCustomTask', ...
    Title = "My Custom Task", ...
    IterationQuery = padv.builtin.query.FindModels);
t.LaunchToolAction=@{openDependencyAnalyzer,@openCloneDetector};
t.LaunchToolText=["Open Dependency Analyzer", "Open Clone Detector"];
```

In this case, `@openDependencyAnalyzer` and `@openCloneDetector` are handles to custom functions that open the Dependency Analyzer app and Clone Detector app, respectively.



Compatibility Considerations

- For `padv.Artifact`, these properties have been removed:
 - Address
 - UUID
 - StorageAddress

To specify or get an artifact address, update your code to use `padv.util.ArtifactAddress` and its properties instead. There is no direct replacement for the properties `UUID` and `StorageAddress`.

- For the `runprocess` function, the `EnableTaskLogging` argument is now `true` by default. Previously, the argument was `logical.empty` by default. Note that if the project setting `SuppressOutputWhenInteractive` is `true` and MATLAB is not running in batch mode, task logging is automatically disabled.

May 2024

Supported releases:

- R2024a
- R2023b
- R2023a
- R2022b Update 1 (and later updates)

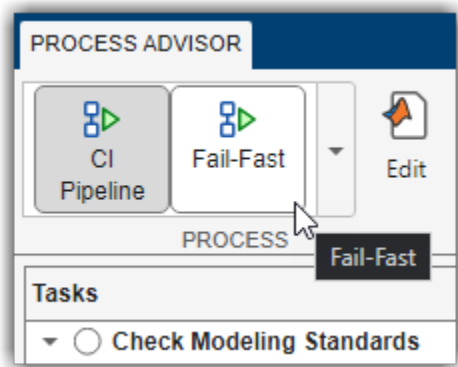
Features

Manage Different Workflows Using Processes

- Inside your process model, you can now define multiple processes for the different build and verification workflows, environments, and other situations that your team needs. For example, you can have one process for your CI pipeline and a separate process for smoke testing with fail-fast tasks. In Process Advisor, you can select which process you want to use from the **Processes** gallery in the toolstrip. APIs like the `runprocess` function also allow you to specify which Process to run.

```
runprocess(Process = "Fail-Fast")  
processadvisor(modelName, "Fail-Fast")
```

If you define multiple processes, use the `padv.Process` methods to add tasks and subprocesses and to specify the relationships within that process. For more information, see “Manage Multiple Build and Verification Workflows Using Processes” on page 2-49.



Performance Optimizations


- Process Advisor now runs performance checks on your process model and generates a warning if tasks in the process model use multiple instances of the same iteration query. You can improve Process Advisor load times by sharing query instances across your process model. For example, if multiple tasks in the process model use the same iteration query, you can update your code to share a single query object instance across these tasks.


Before
<pre>taskA = pm.addTask("taskA", ... IterationQuery = padv.builtin.query.FindModels); taskB = pm.addTask("taskB", ... IterationQuery = padv.builtin.query.FindModels);</pre>
After
<pre>sharedModelsQuery = padv.builtin.query.FindModels(... Name="SharedModelsQuery"); taskA = pm.addTask("taskA", ... IterationQuery = sharedModelsQuery); taskB = pm.addTask("taskB", ... IterationQuery = sharedModelsQuery);</pre>


For more information on how to improve performance, see “Best Practices for Process Model Authoring” on page 2-56. If you want to suppress performance warnings, specify the `padv.ProcessModel` property `EnablePerformanceChecks` as `false` inside your process model.

- Additionally, a query can use the results of another query by specifying that query as a parent. The query can use the parent query to find an initial set of iteration artifacts. You can use the `Parent` name-value argument for these built-in queries:
 - `padv.builtin.query.FindCodeForModel`
 - `padv.builtin.query.FindMAJustificationFileForModel`
 - `padv.builtin.query.FindModelsWithTestCases`
 - `padv.builtin.query.FindRequirementsForModel`
 - `padv.builtin.query.FindTestCasesForModel`

Warnings for Best Practices

- By default, the build system now generates a warning for untracked I/O files. If you make a change to an untracked input or output file, Process Advisor and the build system *do not* mark the task as outdated. Make sure that task inputs or outputs that appear as **Untracked**  do not need to be tracked to maintain the task status and result information that you need for your project.

In Process Advisor, the **I/O** column shows a warning icon  for tasks that have untracked inputs or outputs. To change this behavior, you can specify the project setting **Untracked dependency behavior** as either:

- "Allow" — Do not generate warnings or errors for untracked I/O files.
- "Warn" — Generate a warning if a task has untracked I/O files. In Process Advisor, the **I/O** column shows a warning icon .
- "Error" — Generate an error if a task has untracked I/O files.

For more information, see “Specify Settings for Process Advisor and Build System” on page 1-16.

- You can instruct the build system to detect when there are multiple process model files on the project path. For more information, see the property `DetectMultipleProcessModels` for

`padv.ProjectSettings`. To avoid unexpected behavior, make sure only one `processmodel` file is on the project path.

Built-In Query Enhancements

- Find artifacts where the path matches a regular expression pattern by using the new `IncludePathRegex` and `ExcludePathRegex` name-value arguments for these built-in queries:
 - `padv.builtin.query.FindArtifacts`
 - `padv.builtin.query.FindExternalCodeCache`
 - `padv.builtin.query.FindFilesWithLabel`
 - `padv.builtin.query.FindModels`
 - `padv.builtin.query.FindModelsWithLabel`
 - `padv.builtin.query.FindRequirements`

For example, to find artifacts that start with `DD_` and have an `.sldd` file extension:

```
q = padv.builtin.query.FindArtifacts(...  
IncludePathRegex = "DD_.*\\.sldd");  
run(q)
```

- The built-in query `padv.builtin.query.FindArtifacts` and its subclasses now support Windows[®]-style path separators (`\`) in the paths for `IncludePath` and `ExcludePath`. Previously, the query expected UNIX[®]-style separators (`/`).

April 2024

Supported releases:

- R2024a

Features:

- The support package now supports R2024a.

▲ March 2024

Supported releases:

- R2023b
- R2023a
- R2022b Update 1 (and later updates)

The March 2024 update also makes all features from the February 2024 update available to R2022b, R2023a, and R2023b. See "February 2024" below.

Features

Parallel Code Generation, Integration, and Automation

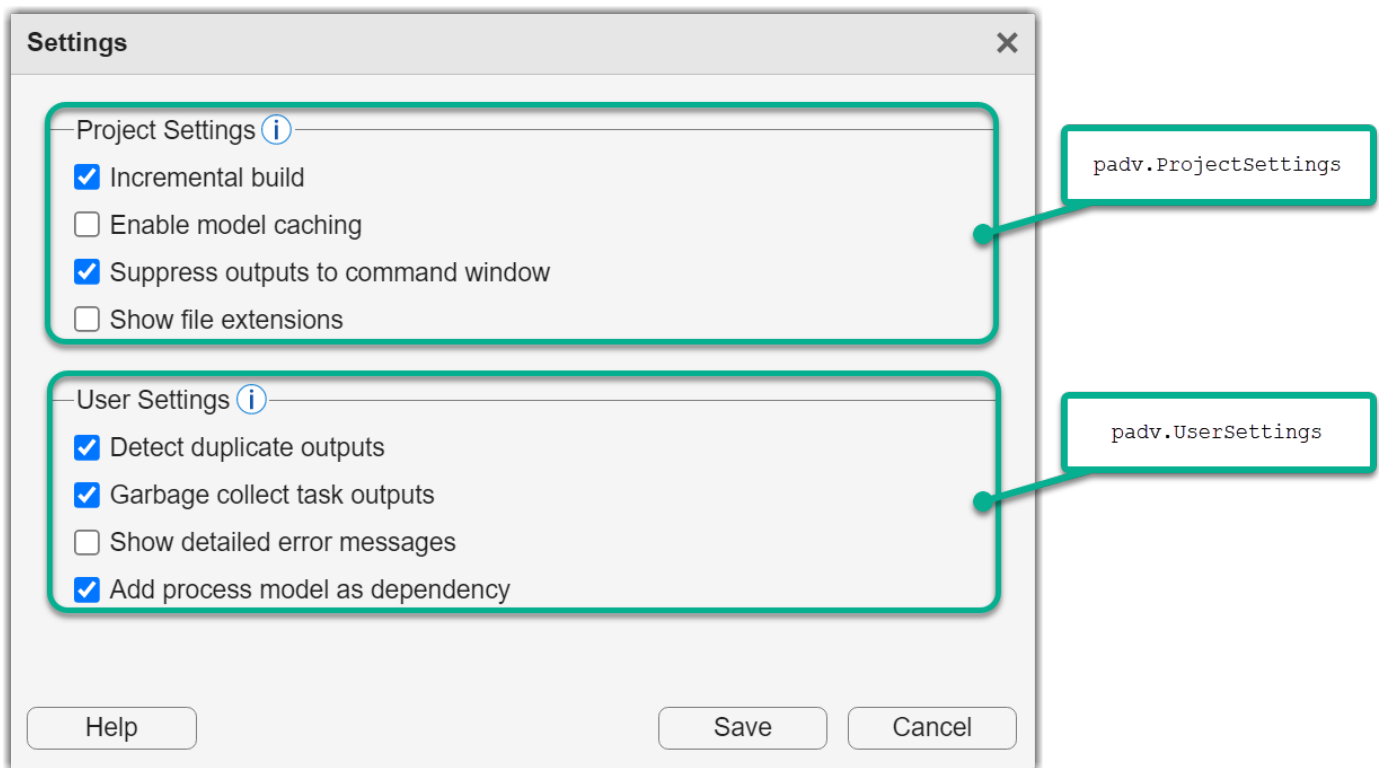
- Starting in R2023b Update 5, the pipeline generator supports a round-trip, parallel CI workflow that automatically merges the task statuses and project analysis from across the parallel branches. Previously, the parallel pipeline architecture `IndependentModelPipelines` generated separate artifact database files, `artifacts.dmr`, for each parallel branch. The pipeline generator uses utility functions to save and merge the artifact database files from parallel branches into a single `artifacts.dmr` file. When you download your CI artifacts onto your machine, you can use the merged `artifacts.dmr` file in your project to see up-to-date task statuses locally in Process Advisor. For information and considerations for parallel code generation, see "Parallel Pipeline Architectures" on page 3-24.
- If you use Git submodules to organize your projects, the pipeline generator, `padv.pipeline.generatePipeline`, now supports automatic fetching of Git submodules for GitHub and GitLab. For more information, see either "Integrate Process into GitHub" on page 3-5 or "Integrate Process into GitLab" on page 3-8.
- Previously, if you wanted to create a Docker image that installed the support package, you needed to download and use the offline installer files. You can now build a Docker image that directly installs the support package and other products using the MATLAB Package Manager (MPM). For information and the updated example Dockerfile, see "Create Docker Container for Support Package" on page 3-30.

Process Advisor Enhancements

- Previously, you used `padv.Preferences` to manage both project and run-time settings. Now, you can specify these settings by using the new classes `padv.ProjectSettings` and `padv.UserSettings`, respectively. These classes allow you to programmatically control the settings for incremental builds, build system logging, and other behaviors.

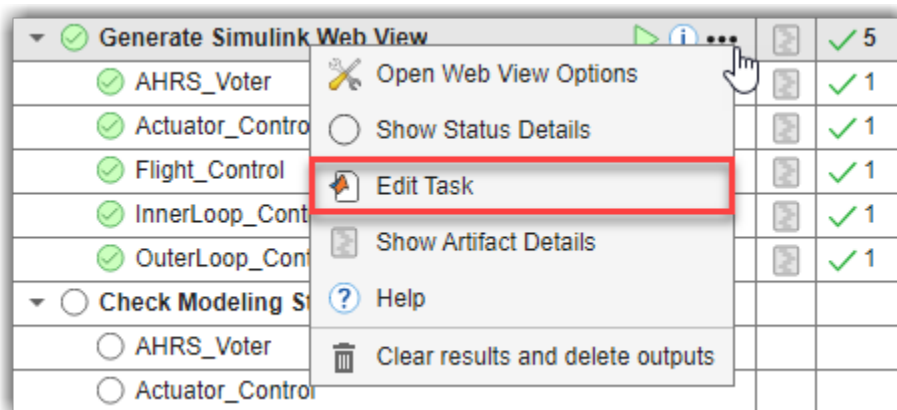
Additionally, you no longer need to create a project startup script to persist run-time settings. The `padv.UserSettings` class automatically manages and persists those settings across MATLAB sessions on your machine.

The main class properties correspond to settings in the Process Advisor Settings dialog box.



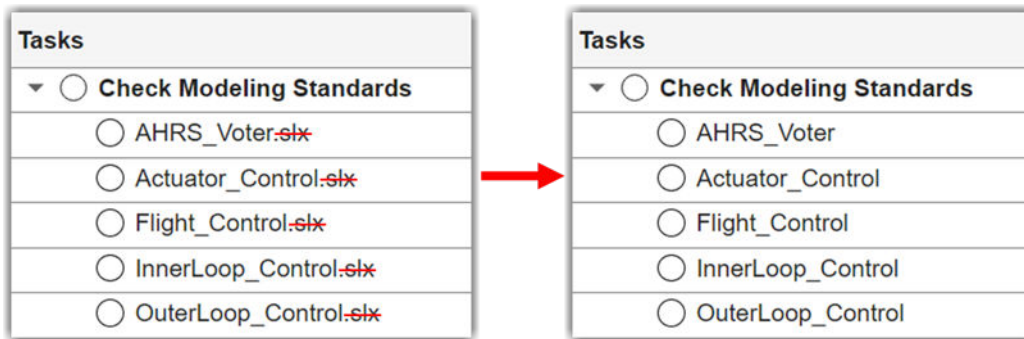
The class `padv.Preferences` will be removed in a future release. For project settings, use `padv.ProjectSettings` instead. For run-time settings, use `padv.UserSettings` instead.

- To view the source code or edit the class definition for a task, you can now point to the task, click the ellipsis (...), and then click **Edit Task**.



For information on how to reconfigure the built-in tasks or create custom tasks, see “Reconfigure Task Behavior” on page 2-17 and “Create Custom Tasks” on page 2-28.

- By default, Process Advisor no longer shows file extensions for task iteration artifacts shown in the **Tasks** column. Previously, you needed to create a custom query if you wanted to remove the file extensions from artifact names in Process Advisor.



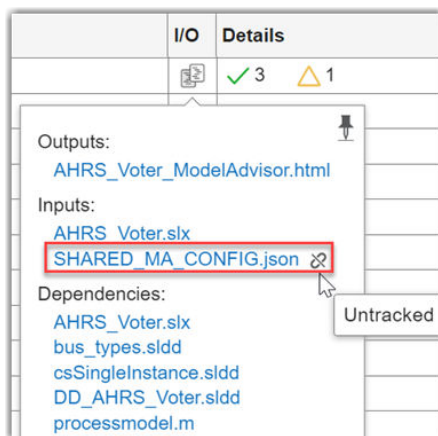
By default, queries now strip file extensions from the `Alias` property of each task iteration artifact. To show file extensions for all artifacts in the **Tasks** column, select the project setting **Show file extensions**. To keep file extensions in the results for a specific query, specify the query property `ShowFileExtension` as `true`.

Build System Enhancements

- You can now use files outside your project as inputs to a task. For example, if you have a shared Model Advisor configuration file, `SHARED_MA_CONFIG.json`, that is outside your project, you can add the file as an input to the **Check Modeling Standards** task.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
maTask.addInputQueries(padv.builtin.query.FindFileWithAddress( ...
    Type='ma_config_file', Path=which('SHARED_MA_CONFIG.json')));
```

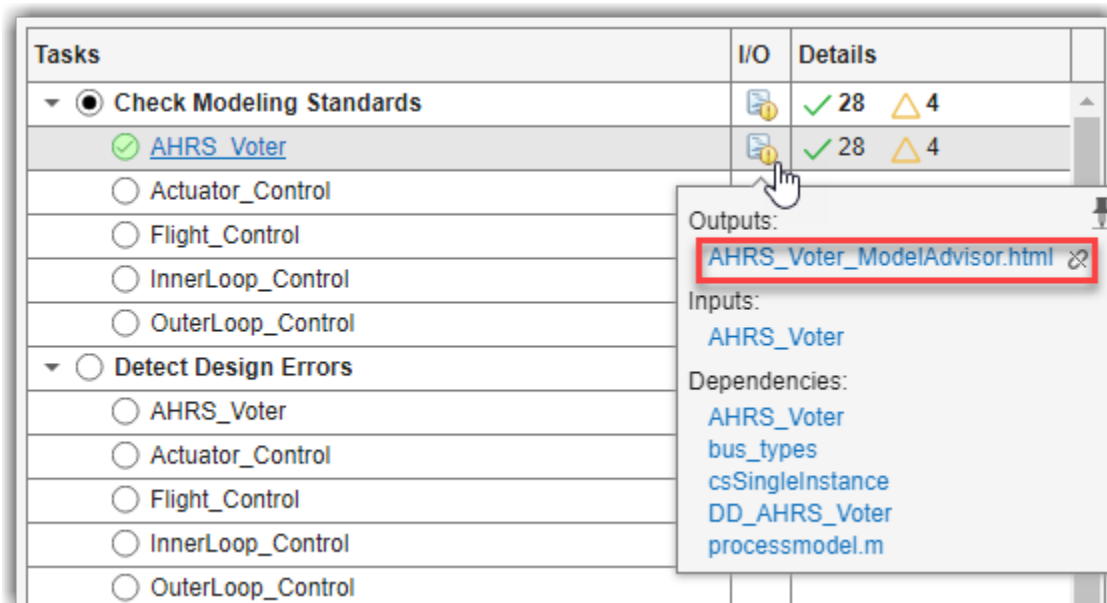
In the Process Advisor **I/O** column, the file appears as **Untracked** because you cannot track changes to files outside the project. If you make a change to an untracked file, the build system does not mark the task as outdated.



- If you do not want the build system to mark a task as outdated when you make changes to task outputs, you can now turn off change tracking for those task outputs. In your process model, specify the task property `TrackOutputs` as `false`.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
maTask.TrackOutputs = false;
```

In the Process Advisor **I/O** column, the outputs appear as **Untracked**. If you make a change to an untracked file, the build system does not mark the task as outdated.



- The build system can now cache requirement sets. For information on caching, see “Cache Models and Other Artifacts Used During Build” on page 3-32.
- You can suppress command-line output from tasks by specifying the new `runprocess` argument `EnableTaskLogging` as `false`. By default, the `runprocess` function only suppresses command-line output from tasks if the project setting `SuppressOutputWhenInteractive` is `true` and MATLAB is not running in batch mode.
- If you want to override the project setting `SuppressOutputWhenInteractive` when you use the function `runprocess` during interactive MATLAB sessions, you can use the `runprocess` argument `SuppressOutputWhenInteractive`. For information, see `runprocess`.

Built-In Tasks and Queries

- The built-in task `padv.builtin.task.AnalyzeModelCode` has been enhanced to:
 - Prevent the task from dirtying the model when you specify a Polyspace configuration options in the process model.
 - Check if MATLAB is already connected to a Polyspace server before calling `polyspaceJobsManager`.
 - Allow you to override the Polyspace configuration options with two new task properties:
 - `Batch` — Option to run analysis on server (`-batch`)
 - `Scheduler` — Specify cluster or job scheduler (`-scheduler`)

For information, see `padv.builtin.task.AnalyzeModelCode`.

- You can use the built-in query `padv.builtin.query.FindCodeForModel` to find the generated code files and `buildInfo.mat` for a model. If you have your code generation tasks and code analysis tasks in different subprocesses, this query can be helpful for passing your generated code other subprocesses. For more information and an example, see the documentation for the built-in query `padv.builtin.query.FindCodeForModel`.
- The following built-in tasks override the model configuration parameter **LaunchReport** to suppress code generation reports from appearing during task execution:
 - `padv.builtin.task.GenerateCode`
 - `padv.builtin.task.RunTestsPerModel`
 - `padv.builtin.task.RunTestsPerTestCase`

Utility Functions

- If you want to manually refresh the process model data, you can use the new utility function `padv.util.refreshProcessModel`. For information, see `padv.util.refreshProcessModel`.
- If you need to get a list of the project references for the current project for a custom task or query, consider using the new utility function `padv.util.getProjectReferences`. This function gets a list of the project references for the current project and caches the list. For information, see `padv.util.getProjectReferences`.

Compatibility Considerations

- The class `padv.Preferences` will be removed in a future release. Update your code to replace instances of `padv.Preferences` with either `padv.UserSettings.get()` or `padv.ProjectSettings.get()`, depending on which property you need to access.

padv.Preferences Property	Update
DetectDuplicateOutputs	Replace instances of <code>padv.Preferences</code> with <code>padv.UserSettings.get()</code> .
GarbageCollectTaskOutputs	
ShowDetailedErrorMessage	
TrackProcessModel	
FilteredDigitalThreadMessages	Replace instances of <code>padv.Preferences</code> with <code>padv.ProjectSettings.get()</code> .
IncrementalBuild	
EnableModelCaching	
MaxNumModelsInCache	
MaxNumTestResultsInCache	
SuppressOutputWhenInteractive	

For example:

Functionality	Use This Instead
<pre>% changing run-time setting p1 = padv.Preferences; p1.DetectDuplicateOutputs = false;</pre>	<pre>p1 = padv.UserSettings.get(); p1.DetectDuplicateOutputs = false;</pre>
<pre>% changing project setting p1 = padv.Preferences; p1.IncrementalBuild = false;</pre>	<pre>p1 = padv.ProjectSettings.get(); p1.IncrementalBuild = false;</pre>

- By default, Process Advisor no longer shows file extensions for artifacts shown in the **Tasks** column. To show file extensions for all artifacts in the **Tasks** column, select the project setting **Show file extension**. To keep file extensions in the results for a specific query, specify the query property `ShowFileExtension` as `true`.

February 2024

February 2024 was released for R2022a Update 4 (and later updates).³

Features

Model and Simulation Management

- Starting in R2023a, you can use your Model Advisor justifications when checking modeling standards. Provide your justification files as inputs to the task by using the new built-in query `padv.builtin.query.FindMAJustificationFileForModel` to find the justification files in a specified folder. For example:

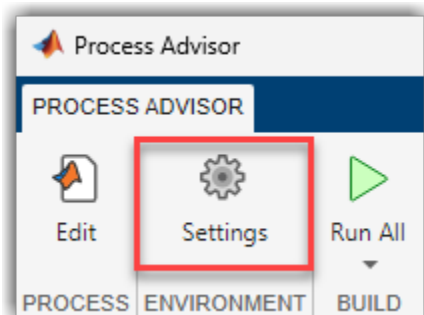
```
maTask = addTask(pm,padv.builtin.task.RunModelStandards);
maTask.addInputQueries(...
    padv.builtin.query.FindMAJustificationFileForModel(...
        JustificationFolder=fullfile("Justifications","ModelAdvisor")));
```

See `padv.builtin.query.FindMAJustificationFileForModel`.

- Starting in R2023a, you can run tests in different simulation modes by specifying the `SimulationMode` property for the built-in tasks `padv.builtin.task.RunTestsPerModel` and `padv.builtin.task.RunTestsPerTestCase`. The property allows you to override the test simulation mode without having to change the test definition.
- The property `DefaultOutputDirectory` for `padv.ProcessModel` now supports paths relative to the project root.

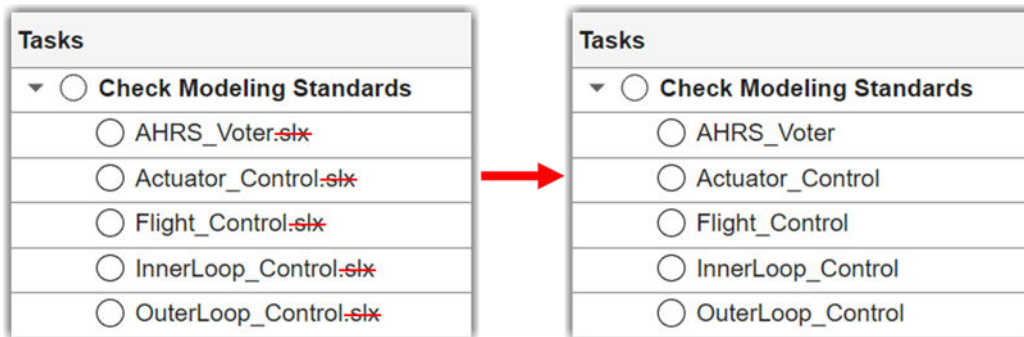
Process Advisor Enhancements

- You can now specify `padv.Preferences` by using the new **Settings** user interface in the Process Advisor.



- You can now customize how artifact names appear in Process Advisor by using the new `Alias` property of `padv.Artifact` objects.

³ The February release is the last planned release for R2022a.



- You can suppress command-line output from Process Advisor during interactive MATLAB sessions by selecting **Suppress outputs to command window** in the Settings dialog box. For information, see “Specify Settings for Process Advisor and Build System” on page 1-16.

Build System Enhancements

- The build system can now run tasks from any working directory. Previously, you needed to be within the project root folder to run tasks.
- Previously during a build, the build system only cached models. Now, when you select the **Enable model caching** setting, the build system can cache models and several other artifacts, including test results, requirements files, and System Composer architecture models. You can control the size of the cache by using the new `padv.Preferences.preferences.MaxNumModelsInCache` and `MaxNumTestResultsInCache`. The built-in tasks now use the new utility function `padv.util.closeModelsLoadedByTask` to close models loaded by the task. For information, see “Cache Models and Other Artifacts Used During Build” on page 3-32.

Utility Function for Custom Tasks and Queries

If you need to get the current project instance for a custom task or query, consider using the new utility function `padv.util.getCurrentProject`. This function can be faster than the `currentProject` function because it creates a persistent variable for the current project instance. For information, see `padv.util.getCurrentProject`.

Compatibility Considerations

Supported Releases

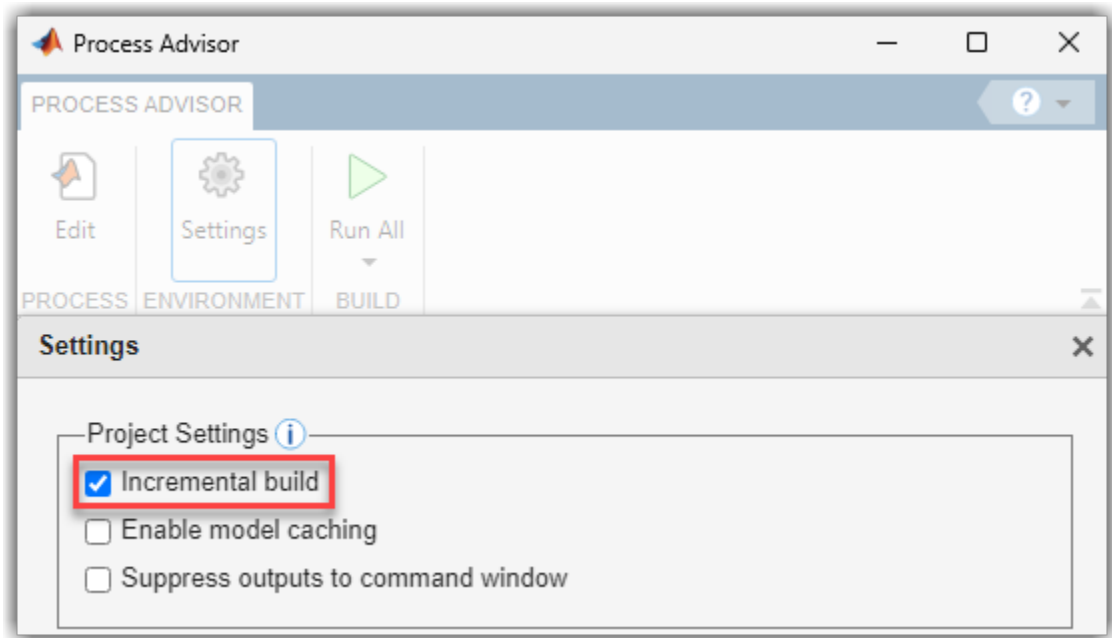
- The February release is the last planned release for R2022a.

In March 2024, the support package will support:

- R2023b
- R2023a
- R2022b Update 1 (and later updates)

Process Advisor

- In Process Advisor, the **Incremental Build** check box is now in the Settings dialog box. In the toolbar, click **Settings** to access the **Incremental build** setting. For information, see “Specify Settings for Builds”.



- **Build System**

- The **Enable model caching** setting (EnableModelCaching property in padv.Preferences) is now off by default.

- **Built-In Tasks**

- The built-in task `padv.builtin.task.RunModelStandards` no longer supports generating reports as PDF files. If you specified the task property `ReportFormat` as "pdf", you must update your code to specify the report format as "html" or "docx" instead.
- For the built-in task `padv.builtin.task.GenerateCode`, the property `IncludeModelReferenceSimulationTargets` has been removed and is no longer supported. Update your code to remove references to `IncludeModelReferenceSimulationTargets`.

- **Artifact Handling**

- The object function `getAlias` has been removed from `padv.Artifact`. To get the human-readable name for an artifact, use the `Alias` property instead.
- The methods `padv.Task.load_model` and `padv.Task.close_model` have been removed and the `padv.Task.load_model` functionality is no longer supported. If you used `padv.Task.load_model` and `padv.Task.close_model` inside your custom tasks, update your code to use a function like `load_system` to load your model and use the new utility function `padv.util.closeModelsLoadedByTask` to close the models loaded by a task. For information, see `padv.util.closeModelsLoadedByTask`.

▲ December 2023

Supports:

- R2023b
- R2023a
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

- Starting in R2023b Update 5, you can merge `artifacts.dmr` files from different branches or CI jobs to make sure task statuses are up-to-date with the latest project analysis.

- Save a copy of an artifact database file using the function `padv.util.saveArtifactDatabase`:

```
padv.util.saveArtifactDatabase(fullfile("derived", "base.dmr"))
```

- Merge artifact database files using the function `padv.util.mergeArtifactDatabases`:

```
padv.util.mergeArtifactDatabases(...
Base = fullfile("derived", "base.dmr"), ...
Branches = [fullfile("derived", "featureA.dmr"), fullfile("derived", "featureB.dmr)], ...
Merged = fullfile("derived", "artifacts.dmr"))
```

The merged `artifacts.dmr` file contains the updates from the specified branches.

- You can improve the efficiency of model loading in your builds by using the methods `padv.Task.load_model` and `padv.Task.close_model` inside your custom tasks. These methods allow the build system to cache a model, instead of reloading the same model multiple times within a build. For information, see "Best Practices for Effective Builds".
- The built-in task `padv.builtin.task.MergeTestResults` can generate code coverage reports for tests that you execute in software-in-the-loop (SIL) mode and processor-in-the-loop (PIL) mode. The report names are specified by the new task properties `CovReportNameSIL` and `CovReportNamePIL`. For more information, see the documentation for the built-in task `padv.builtin.task.MergeTestResults`.
- Programmatically get task results from specific tasks, subprocesses, and artifacts by using the name-value arguments for the function `getProcessTaskResults`. For example, to get the task results from running the task `padv.builtin.task.RunModelStandards` on the artifact `myModel.slx`:

```
[IDsWithResults, results, outdated] = getProcessTaskResults(...
Tasks = "padv.builtin.task.RunModelStandards", ...
FilterArtifact = fullfile("models", "myModel.slx"))
```

For information, see `getProcessTaskResults`.

- You can get the outputs from a specific task by using the `Task` argument for the built-in query `padv.builtin.query.GetOutputsOfDependentTask`. You can also specify a unique query name using the `Name` argument. For example:

```
padv.builtin.query.GetOutputsOfDependentTask(...  
Task="padv.builtin.task.GenerateCode",...  
Name = "CustomNameForQuery")
```

For information, see `padv.builtin.query.GetOutputsOfDependentTask`.

- When you use the pipeline generator, you no longer need to specify the `OutputDirectory` property for custom tasks. If your custom task generates outputs without a specified output directory, the build system automatically stores the task outputs in the `DefaultOutputDirectory` specified in the process model.
- If you want to filter out certain types of issues shown in the **Project Analysis Issues** pane, you can use the `FilteredDigitalThreadMessages` in your `padv.Preferences`. For information, see `padv.Preferences`.

⚠ Compatibility Considerations

- Built-in tasks now use the methods `padv.Task.load_model` and `padv.Task.close_model` to improve the efficiency builds by caching models. If you do not want tasks to cache models, specify the `EnableModelCaching` property in your `padv.Preferences` as `false`.

▲ November 2023

Supports:

- R2023b
- R2023a
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

- Check for run-time errors in every operation in your code by configuring the built-in task `padv.builtin.task.AnalyzeModelCode` to use Polyspace Code Prover.

When you specify the task property `VerificationMode` as "CodeProver", the task uses Polyspace Code Prover to prove code quality.

You can use both Bug Finder and Code Prover in your software development workflow. To include both a Bug Finder task and a Code Prover task in your process model, add two separate instances of the built-in task `padv.builtin.task.AnalyzeModelCode` to your process. For example:

```
%% Check Coding Standards with Polyspace Bug Finder
psbfTask = pm.addTask(padv.builtin.task.AnalyzeModelCode());
% Report Options
psbfTask.ResultDir = fullfile(defaultResultPath, 'bug_finder');
psbfTask.ReportPath = fullfile(defaultResultPath, 'bug_finder');

%% Prove Code Quality with Polyspace Code Prover
pscpTask = pm.addTask(padv.builtin.task.AnalyzeModelCode(Name="ProveCodeQuality"));
pscpTask.Title = "Prove Code Quality";
pscpTask.VerificationMode = "CodeProver";
% Report Options
pscpTask.ResultDir = string(fullfile(defaultResultPath, 'code_prover'));
pscpTask.Reports = ["Developer", "CallHierarchy", "VariableAccess"];
pscpTask.ReportPath = string(fullfile(defaultResultPath, 'code_prover'));
pscpTask.ReportNames = [...
    "$ITERATIONARTIFACT$ _Developer", ...
    "$ITERATIONARTIFACT$ _CallHierarchy", ...
    "$ITERATIONARTIFACT$ _VariableAccess"];
```

For more information, see the documentation for the built-in task `padv.builtin.task.AnalyzeModelCode`.

- Find multiple files with the built-in query `padv.builtin.query.FindFileWithAddress` by specifying the artifact type and file path name-value arguments as vectors of the same length.

```
padv.builtin.query.FindFileWithAddress(...
    Type=[artifactType1, artifactType2],...
    Path=[filePath1, filePath2])
```

For more information, see the documentation for the built-in query `padv.builtin.query.FindFileWithAddress`.

(continues on next page)

- By default, the build system now generates an error if multiple tasks attempt to write to the same output file. If you want to turn this setting off, you can specify `DetectDuplicateOutputs` as `false` in `padv.Preferences`.
- The built-in query `padv.builtin.query.FindTestCasesForModel` can now also find test cases associated with subsystem references. A subsystem reference allows you to save the contents of a subsystem in a separate file and reference it using a Subsystem Reference block. Previously, the query found only the test cases directly associated with the Simulink or System Composer model itself.

Fixes:

- A syntax issue has been fixed in the example pipeline configuration file for GitLab. You can open the updated example by entering `processAdvisorGitLabExampleStart` in the MATLAB Command Window.

⚠ Compatibility Considerations

- In a future release, the built-in query `padv.builtin.query.FindFileWithAddress` will no longer accept positional arguments. Update your code to use name-value arguments instead.

Functionality	Use This Instead
<pre>padv.builtin.query.FindFileWithAddress(... "artifactType",... "filePath")</pre>	<pre>padv.builtin.query.FindFileWithAddress(... Type = "artifactType",... Path = "filePath")</pre>

October 2023

Supports:

- R2023b
- R2023a
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features

- You can compare models to their ancestors in Git and generate a model comparison report directly from Process Advisor with the built-in task `padv.builtin.task.GenerateModelComparison`.

To add the task to your process model, use the function `addTask`:

```
mdlCompTask = addTask(pm, padv.builtin.task.GenerateModelComparison());
```

You can use the task properties to specify different report options, filtering options, and the name of the Git branch used for the comparison. For example:

```
mdlCompTask.ReportFormat = "DOCX";
mdlCompTask.MainBranch = "branchname";
```

In Process Advisor, when you point to the task and click **... > Compare to Ancestor**, you can open the Model Comparison tool.

Tasks	I/O	Details
<ul style="list-style-type: none"> Generate Model Comparison AHRS_Voter.slx Actuator_Control.slx Flight_Control.slx InnerLoop_Control.slx OuterLoop_Control.slx 	<ul style="list-style-type: none"> 5 1 1 1 1 1 	<ul style="list-style-type: none"> 5 1 1 1 1 1
<ul style="list-style-type: none"> Generate SDD AHRS_Vote Actuator_Co Flight_Control.slx 		

For more information, see the documentation for the built-in task `padv.builtin.task.GenerateModelComparison`.

(continues on next page)

Note If you run MATLAB using the `-nodisplay` option or you use a machine that does not have a display (like many CI runners and Docker containers), you should set up a virtual display server before you include this task in your process model. For information, see “Set Up Virtual Display Machines Without Displays” on page 3-29.

- By default, the built-in query `padv.builtin.query.FindFileWithAddress` validates that the file exists before returning the file from the query. The name-value argument `ValidateFileExistence` is now `true` by default.

September 2023

Supports:

- R2023a
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features

- Manually generate a pipeline configuration file for GitHub by passing a `padv.pipeline.GitHubOptions` object to the function `padv.pipeline.generatePipeline`. For information, see “Integrate Process into GitHub” on page 3-5.
- Group related tasks, create a hierarchy of tasks, and share parts of a process using subprocesses. A *subprocess* is a self-contained sequence of tasks, inside a process or other subprocess, that can run standalone. For information, see “Group Tasks Using Subprocesses”.

Tasks	I/O	Details
<input type="radio"/> Task 1		
▼ <input type="radio"/> Subprocess A		
<input type="radio"/> Task A1		
<input type="radio"/> Task A2		
▼ <input type="radio"/> Subprocess B		
<input type="radio"/> Task B1		
<input type="radio"/> Task B2		

- Programmatically run tasks, subprocesses, and tasks for specific artifacts by using the updated name-value arguments for the `runprocess` function:
 - Tasks — Specify the names of the tasks that you want to run.


```
runprocess(...
  Tasks = ["padv.builtin.task.GenerateSimulinkWebView", ...
  "padv.builtin.task.RunModelStandards"])
```
 - Subprocesses — Specify the name of the subprocess that you want to run.

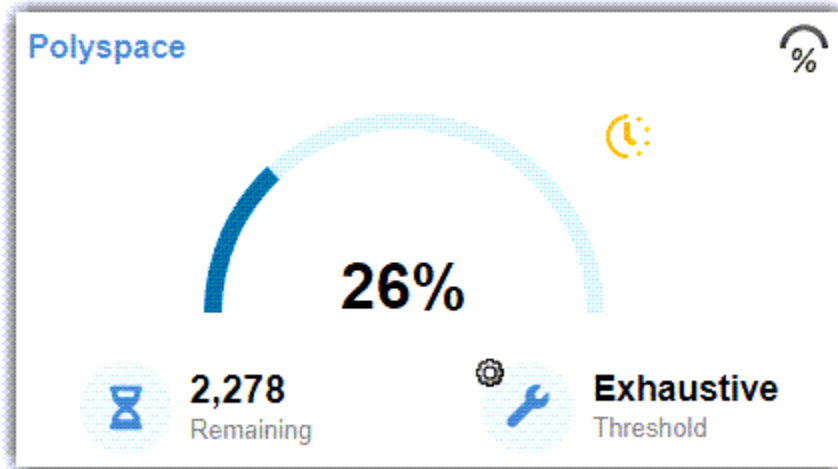

```
runprocess(Subprocesses = "SubprocessA")
```
 - FilterArtifact — Specify the artifact that you want to run tasks on.


```
runprocess(...
  FilterArtifact = fullfile("02_Models", "AHRV_Voter", "specification", "AHRV_Voter.slx"))
```

You can use one or more of these name-value arguments to specify what you want to run. You can also use these name-value arguments with the function `generateProcessTasks`. For more information, see `runprocess` and `generateProcessTasks`.

(continues on next page)

- You can reconfigure the **Check Coding Standards** task (`padv.builtin.task.AnalyzeModelCode`) to automatically upload Bug Finder analysis results to Polyspace Access.



Use the new Polyspace Access properties of the task to provide your configuration options and credentials. For example, for a process model with a Polyspace task object `psTask`:

```
% Polyspace Access configuration options
psTask.PsAccessEnable = true;
psTask.PsAccessHostName = "my-polyspace-access";
psTask.PsAccessPortNumber = "9443";
psTask.PsAccessProtocol = "https";
psTask.PsAccessCredentialsFile = "C:\Users\username\myCredentials.txt";
psTask.PsAccessParentFolder = "public/myProject";
psTask.PsAccessResultsName = "$ITERATIONARTIFACT$_CodingStandards";
```

For more information, see the documentation for the built-in task `padv.builtin.task.AnalyzeModelCode`.

- By default, a query can find files under the project root folder, even if you did not add that file to the project. To only return artifacts that you added to the project, you can now specify the `InProject` argument for the query as `true`.

For example, to have the **Check Modeling Standards** task, `maTask`, only run for models that you added to the project, specify the iteration query as `padv.builtin.query.FindModels` and specify the argument `InProject` as `true`.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
maTask.IterationQuery = padv.builtin.query.FindModels(...
    InProject = true);
```

The `InProject` argument is available for the built-in queries `FindArtifacts`, `FindFilesWithLabel`, `FindModels`, `FindModelsWithLabel`, and `FindRequirements`.

- When you open a test case from the **Tasks** column, Process Advisor automatically loads the test case results in Test Manager.

▲ August 2023

Supports:

- R2023b
- R2023a
- R2022b Update 1 (and later updates)

Features

- With the pipeline generator, you can run tasks on your models in parallel by using the pipeline architecture, `padv.pipeline.Architecture.IndependentModelPipelines`. Downstream, parallel pipelines independently run the tasks associated with each model. For more information, see "Integrate into CI".

Fixes

- Previously, if you set the properties of a query instance in the process model, all tasks that used that query instance were affected, unless you specified a unique name for the query instance. Now, you no longer need to specify a unique name for the query instance to set different values for different tasks. For example, you can have two tasks, TaskA and TaskB, that set different properties for instances of the built-in query `padv.builtin.query.FindModels`.

```
% Task A only runs on the model "A.slx"
taskA = addTask(pm, "TaskA");
taskA.IterationQuery = padv.builtin.query.FindModels;
taskA.IterationQuery.IncludePath = "A.slx";
```

```
% Task B only runs on the model "B.slx"
taskB = addTask(pm, "TaskB");
taskB.IterationQuery = padv.builtin.query.FindModels;
taskB.IterationQuery.IncludePath = "B.slx";
```

If you want to share a query across multiple tasks, specify a unique name for the query and use the `addQuery` function to add the query to the process model.

- The build system no longer returns a warning or exception when attempting to load results generated by a previous version of the support package.

▲ Compatibility Considerations

- You must specify the Name property for a query instance before you use the `addQuery` function in the process model.

July 2023

Supports:

- R2023a
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Fixes

- Removed unsupported call to `padv.utils.isMACacheUpdated` in the built-in task `padv.builtin.task.RunModelStandards` (**Check Modeling Standards**).

Features:

- The built-in tasks `padv.builtin.task.RunTestsPerModel` and `padv.builtin.task.RunTestsPerTestCase` support test cases that run test iterations in fast restart.
- The built-in task `padv.builtin.task.MergeTestResults` has a new property `LoadSimulationSignalData`. If you specify `LoadSimulationSignalData` as `true`, the task loads simulation signal data when loading the test results.

▲ June 2023

Supports:

- R2023a
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

• Artifacts

- There are new utility functions for working with artifacts. For information, enter:

```
help padv.util
```

- You can use the utility functions when working with artifacts and artifact addresses. For example, you can use `padv.util.ArtifactAddress` to specify the address of a `padv.Artifact`:

```
model = padv.Artifact("sl_model_file", ...
    padv.util.ArtifactAddress(...
    fullfile("02_Models", "AHRV_Voter", "specification", "AHRV_Voter.slx")));
```

• Build System

- You can automatically generate a build report after running tasks with `runprocess`:

```
runprocess(GenerateReport = true)
```

For information on how to specify a different report name and format, see "Generate Build Report".

- Process Advisor and the build system support a P-coded process model file `processmodel.p`. If you have both a P-code file and a `.m` file, the P-code file takes precedence over the corresponding `.m` file for execution, even after modifications to the `.m` file.

• Built-In Tasks and Queries

- You can use the `Tags` argument of the built-in query `padv.builtin.query.FindTestCasesForModel` to find test cases that use specific tags.
- The built-in tasks `padv.builtin.task.RunTestsPerModel` and `padv.builtin.task.RunTestsPerTestCase` now use the MATLAB test runner, `matlab.unittest.TestRunner`, to run tests and generate JUnit-style XML reports in CI.

• Pipeline Generation

- The pipeline generator now allows you to specify if and when you want to collect artifacts for your pipeline. In `padv.pipeline.GitLabOptions` or `padv.pipeline.JenkinsOptions`, you can specify the property `EnableArtifactCollection` as:

- "never", 0, or false — Never collect artifacts
- "on_success" — Only collect artifacts when the job succeeds
- "on_failure" — Only collect artifacts when the job fails
- "always", 1, or true — Always collect artifacts

(continues on next page)

- The pipeline generator now allows you to control whether a pipeline stops or continues running after a stage fails. In `padv.pipeline.GitLabOptions` or `padv.pipeline.JenkinsOptions`, you can specify the property `StopOnStageFailure` as either `true` or `false`. By default, the pipeline does not stop if a stage in the pipeline fails.
- The pipeline generator automatically generates a Process Advisor build report before collecting build artifacts. The report generates in a new job, `Generate_PADV_Report`. For more information, see “How Pipeline Generation Works” on page 3-21.

▲ Compatibility Considerations

• Artifacts

- `padv.Artifact` no longer returns the properties `Address`, `UUID`, `Label`, and `StorageAddress`. `padv.Artifact` now returns an `ArtifactAddress` property instead:

```
a =
```

```
Artifact with properties:
```

```
    Type: "artifact_type"
    Parent: [0x0 padv.Artifact]
    ArtifactAddress: [1x1 padv.util.ArtifactAddress]
```

For information, see the documentation for the utility function `padv.util.ArtifactAddress`.

• Queries

- The `Name` property for `padv.Query` objects is now immutable. You cannot change the value of the `Name` property after the query object is created. If you want to set a property value for a `padv.Query` object, set the value by using the name-value arguments in the constructor.

• Built-In Tasks and Queries

- The `CovReportPath` property was removed from the built-in task `padv.builtin.task.MergeTestResults`. The coverage and test reports automatically generate into the folder location specified by `ReportPath`.
- The `Tags` property was removed from the built-in task `padv.builtin.task.RunTestsPerTestCase`. Use `Tags` argument of query `padv.builtin.query.FindTestCasesForModel` to find test cases with specific test tags instead:

```
addTask(pm, padv.builtin.task.RunTestsPerTestCase, ...
    IterationQuery = padv.builtin.query.FindTestCasesForModel(...
    Tags="FeatureA"));
```

- The `Tags` property will be removed from the built-in task `padv.builtin.task.RunTestsPerModel` in a future release. Use the `Tags` argument of query `padv.builtin.query.FindTestCasesForModel` instead.
- The `GenerateJUnitForTask` property was removed from `padv.Task`. `padv.Task` now uses the properties `CISupportOutputsForTask` and `CISupportOutputsByTask` to control whether tasks generate CI aware result files, like JUnit-style XML reports.
- The built-in tasks `padv.builtin.task.RunTestsPerModel` and `padv.builtin.task.RunTestsPerTestCase` no longer support test cases that run test iterations in fast restart.

- **Pipeline Generation**

- The property `ArtifactsWhen` will be removed from `padv.pipeline.GitLabOptions` in a future release. Use the property `EnableArtifactCollection` to specify when artifacts are collected instead.

(continues on next page)

- The property `SaveArtifactsOnSuccess` will be removed from `padv.pipeline.JenkinsOptions` in a future release. Use the property `EnableArtifactCollection` to specify when artifacts are collected instead.

▲ April 2023

Supports:

- R2023a
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

- The pipeline generator automatically generates JUnit-style XML reports for tasks. The JUnit reports allow you to see a summary of task results directly in the GitLab or Jenkins user interface. For information, see “Integrate Process into GitLab” on page 3-8 or “Integrate Process into Jenkins” on page 3-14.
- The support package contains an example `Dockerfile` for creating a Docker container to run MATLAB with the support package and other MathWorks products. For more information, see “Create Docker Container for Support Package” on page 3-30.
- `padv.ProcessModel` has a property `DefaultOutputDirectory` which controls the `$DEFAULTOUTPUTDIR$` token in the example `processmodel.m` file. By default, Process Advisor outputs files inside a `PA_Results` folder in the project root.
- You can filter the artifacts returned by built-in queries like `padv.builtin.query.FindCodeFolderForModel` by using the properties `IncludeLabel`, `ExcludeLabel`, `IncludePath`, and `ExcludePath`.

```
q = padv.builtin.query.FindRequirements(...
ExcludePath = "HighLevel");
run(q)
```

- The task `padv.builtin.task.MergeTestResults` now supports inputs that supply multiple test results and supports dependencies on multiple predecessor tasks.

▲ Compatibility Considerations

- Previously, several built-in tasks ran on either reference models (**Ref**) or top models (**Top**). These tasks have been combined into a single task that can automatically run on both reference models and top models:

Previous Built-In Task Name	Current Built-In Task Name
<code>padv.builtin.task.AnalyzeRefModelCode</code>	<code>padv.builtin.task.AnalyzeModelCode</code>
<code>padv.builtin.task.AnalyzeTopModelCode</code>	
<code>padv.builtin.task.GenerateCodeAsRefModel</code>	<code>padv.builtin.task.GenerateCode</code>
<code>padv.builtin.task.GenerateCodeAsTopModel</code>	
<code>padv.builtin.task.RunCodeInspectionAsRefModel</code>	<code>padv.builtin.task.RunCodeInspection</code>
<code>padv.builtin.task.RunCodeInspectionAsTopModel</code>	

(continues on next page)

Update your code to use the current built-in task names or instances.

```
% Using current built-in task instances
psTask = pm.addTask(padv.builtin.task.AnalyzeModelCode());
codegenTask = pm.addTask(padv.builtin.task.GenerateCode());
slciTask = pm.addTask(padv.builtin.task.RunCodeInspection());
```

If you want the task to only run on either reference models or top models, you can use the properties of the task (`TreatAsRefModel` or `IsTopModel`) to override the default behavior. For example:

```
% To override the default behavior

psRefTask = pm.addTask(padv.builtin.task.AnalyzeModelCode(...
    TreatAsRefModel = true,...
    IterationQuery = padv.builtin.query.FindRefModels));

codegenRefMdlTask = pm.addTask(padv.builtin.task.GenerateCode(...
    TreatAsRefModel = true,...
    IterationQuery = padv.builtin.query.FindRefModels));

slciRefTask = pm.addTask(padv.builtin.task.RunCodeInspection(...
    IsTopModel = false,...
    IterationQuery = padv.builtin.query.FindRefModels));
```

If your process model uses multiple instances of a task, like `padv.builtin.task.RunCodeInspection`, make sure to specify a unique `Name` for each instance of the task.

```
% Provide unique names

slciTopTask = pm.addTask(padv.builtin.task.RunCodeInspection(...
    Name = "inspectCodeTop",...
    Title = "Inspect Code (Top)",...
    IsTopModel = true,...
    IterationQuery = padv.builtin.query.FindTopModels));

slciRefTask = pm.addTask(padv.builtin.task.RunCodeInspection(...
    Name = "inspectCodeRef",...
    Title = "Inspect Code (Ref)",...
    IsTopModel = false,...
    IterationQuery = padv.builtin.query.FindRefModels));
```

- The options structures, `RunOptions` and `ReportOptions`, for built-in tasks will be removed in a future release. The options structures have been replaced by properties of the built-in tasks. To reconfigure a built-in task, use the properties of the task instead.

For example:

Previously	Now
<code>maTask.RunOptions.ReportPath</code>	<code>maTask.ReportPath</code>

You can open the source code for a built-in task to see a mapping of the options structure to the task properties. For example:

```
open padv.builtin.task.RunModelStandards
```

The getLegacyOptions function shows the mapping. For example:

```
function options = getLegacyOptions()
options = [ ...
    "RunOptions.CheckIDList", "CheckIDList" ...
    "RunOptions.DisplayResults", "DisplayResults"...
    "RunOptions.Force", "Force" ...
    "RunOptions.ParallelMode", "ParallelMode" ...
    "RunOptions.TempDir", "TempDir" ...
    "RunOptions.ShowExclusions", "ShowExclusions" ...
    "RunOptions.ExtensiveAnalysis", "ExtensiveAnalysis" ...
    "RunOptions.ReportName", "ReportName" ...
    "RunOptions.ReportFormat", "ReportFormat" ...
    "RunOptions.ReportPath", "ReportPath" ...
];
end
```

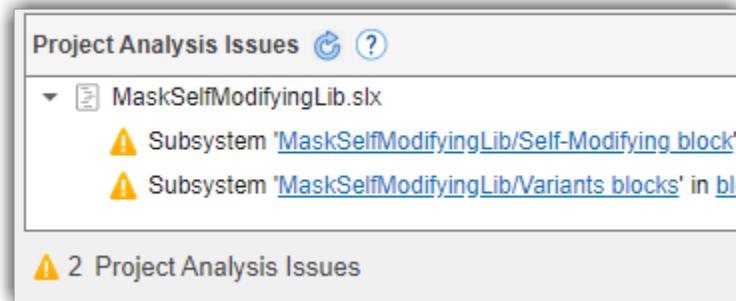
March 2023

Supports:

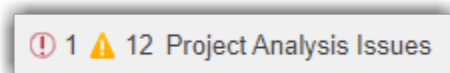
- R2023a
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

- The support package now supports R2023a.
- Starting in R2023a:
 - The support package can analyze artifacts in referenced projects.
 - The **Project Analysis Issues** pane returns warnings for artifacts in the project.



The number of errors and warnings in the project are summarized at the bottom of the Process Advisor app.



▲ February 2023

Supports:

- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

- Automatically generate a pipeline file for a Jenkins pipeline by using the function `padv.pipeline.generatePipeline`. For more information, see “Integrate Process into Jenkins” on page 3-14.
- The CI options for pipeline generation have two new properties:
 - `AddBatchStartupOption` — Specify whether to open MATLAB using the `-batch` startup option
 - `GeneratedPipelineDirectory` — Specify where the generated pipeline file generates
- `padv.Task` has new properties:
 - `AlwaysRun` — If you specify `AlwaysRun` as `true`, the task will always run, even if the task results are already up to date.
 - `LaunchToolText` — Specify a tooltip for a custom launch action for a task.
 - `OutputDirectory` — Location for standard outputs that the task produces
 - `CacheDirectory` — Location for additional cache files that the task generates
- The built-in query `padv.builtin.query.FindArtifacts` accepts a cell array of multiple artifact types for the `ArgumentType` argument. For example, to find the Simulink models and MATLAB M files in a project:

```
q = padv.builtin.query.FindArtifacts(...  
ArtifactType={"sl_model_file", "m_file"});  
run(q)
```

Fixes:

- In the standalone Process Advisor window, Linux users can point to a task and click the ellipses (...) without having to use the arrows on the keyboard to interact with the options in the menu.

▲ Compatibility Considerations

- The `ArtifactsPath` property was removed from `padv.pipeline.GitLabOptions` and `padv.pipeline.JenkinsOptions`. If you previously specified the `ArtifactsPath` property, update your code to no longer specify `ArtifactsPath`. The pipeline generator uses the `OutputDirectory` property of the task to automatically identify which artifacts to collect.

December 2022

Supports:

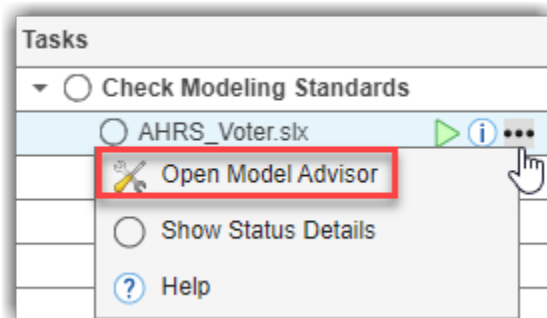
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

- Automatically generate a pipeline configuration file for a GitLab pipeline by using the new function `padv.pipeline.generatePipeline`. For more information, see “Integrate Process into GitLab” on page 3-8 or enter:

help `padv.pipeline.generatePipeline`

- Open the tool associated with a task by pointing to the task in the Process Advisor app and clicking the ellipsis (...) and then **Open Tool Name**.



- Automatically view detailed statuses, inputs, outputs, and dependencies for tasks and task results shown in the Process Advisor app.
- The built-in task **Design Error Detection** now outputs the Simulink Design Verifier data file as an output in the **I/O** column.
- Find artifacts in your project that meet specific search criteria by using the new built-in query `padv.builtin.query.FindArtifacts`.

For information, enter:

help `padv.builtin.query.FindArtifacts`

- Find requirement sets in your project and requirement links to models by using the new built-in queries `padv.builtin.query.FindRequirements` and `padv.builtin.query.FindRequirementsForModel`, respectively.

November 2022

Supports:

- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

- You can now open artifacts, in their associated tool, directly from the Process Advisor app. In the **Tasks** column, point to the name of an artifact and click the hyperlink.
- When there is a new version of the support package available, the Process Advisor app shows an update icon in the bottom-right corner.
- The built-in task for generating a Simulink Web view now includes additional options like the ability to include user notes and export models in subfolders. To view the source code for the task, enter this code in the MATLAB Command Window:

```
open padv.builtin.task.GenerateSimulinkWebView
```

Fixes:

- The Process Advisor app respects requests to cancel artifact analysis.
- The task `padv.builtin.task.AnalyzeModelCode` returns an error if Polyspace Bug Finder is either not installed or not linked to the current MATLAB installation.

October 2022

Supports:

- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

- The support package now supports R2022b for Update 1 and later updates.
- Turn off incremental builds for a project by clearing the **Incremental Build** check box in the Process Advisor app.
- The build system and Process Advisor app take advantage of `runsAfter` relationships when determining the task execution order for tasks associated with the project.

September 2022

Supports:

- R2022a Update 4 (and later updates)

Features:

- You can create a new example project instance that includes an example YAML file for configuring GitLab pipelines:

`processAdvisorGitLabExampleStart`

The example YAML file, `.gitlab-ci.yml`, is in the project root.

- You can create a new example project instance that includes an example Jenkinsfile for configuring Jenkins pipelines:

`processAdvisorJenkinsExampleStart`

The example Jenkinsfile, `Jenkinsfile`, is in the project root.

- Test harnesses are now tracked as dependencies for test cases.
- Externally-saved input or output baselines (including `.mat` and Excel) are now tracked as dependencies for test cases.

Fixes:

- If you are using the project window and there is an error, the error dialog is able to open the artifact listed in the hyperlink.

August 2022

Initial release.

Supports:

- R2022a Update 4 (and later updates)

