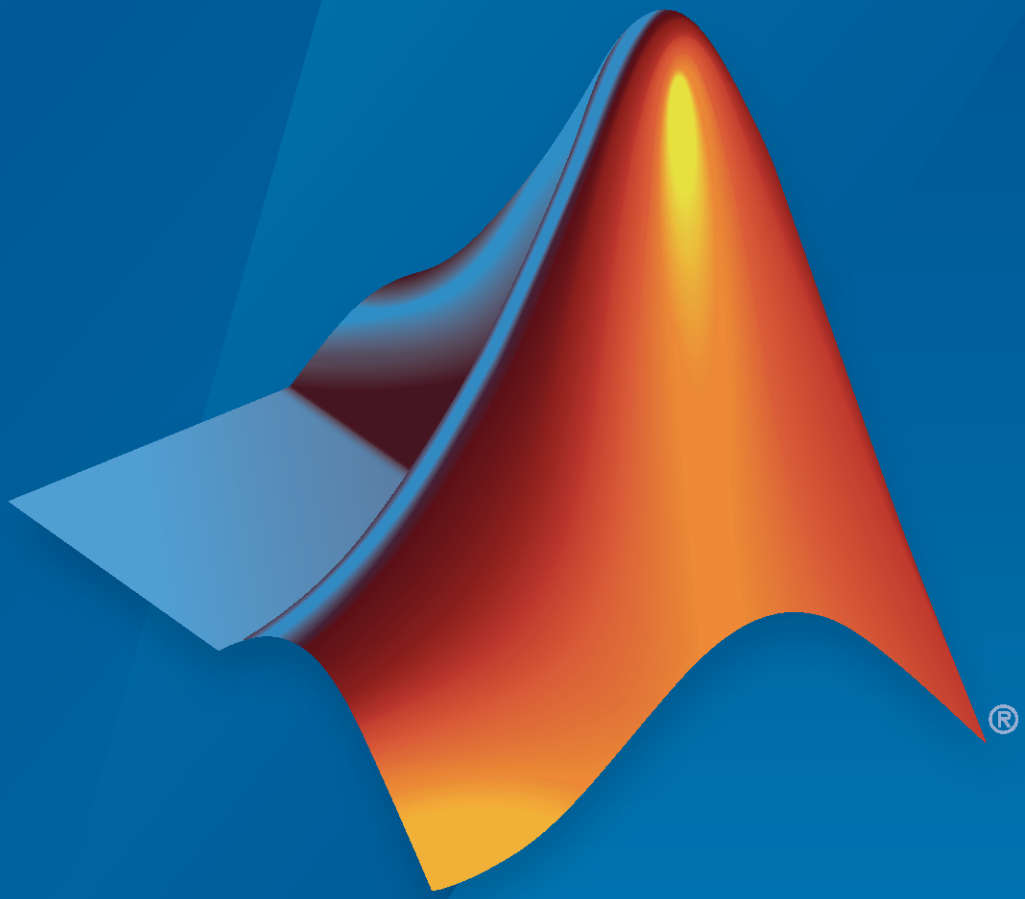
CI/CD Automation for Simulink® Check™ Support Package

User's Guide



MATLAB® & SIMULINK®

R2022b — R2024a

MathWorks®

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

Revision History

August 2022	PDF Only	Version 22.1.0 (R2022a)
September 2022	PDF Only	Version 22.1.1
October 2022	PDF Only	Versions 22.1.2 and 22.2.2 (R2022b)
November 2022	PDF Only	Versions 22.1.3 and 22.2.3
December 2022	PDF Only	Versions 22.1.4 and 22.2.4
February 2023	PDF Only	Versions 22.1.5 and 22.2.5
March 2023	PDF Only	Version 23.1.5 (R2023a)
April 2023	PDF Only	Versions 22.1.6, 22.2.6, and 23.1.6
June 2023	PDF Only	Versions 22.1.7, 22.2.7, and 23.1.7
July 2023	PDF Only	Versions 22.1.8, 22.2.8, and 23.1.8
August 2023	PDF Only	Versions 22.2.9, 23.1.9, and 23.2.0 (R2023b)
September 2023	PDF Only	Versions 22.1.9, 22.2.10, and 23.1.10
October 2023	PDF Only	Versions 22.1.10, 22.2.11, 23.1.11, and 23.2.1
November 2023	PDF Only	Versions 22.1.11, 22.2.12, 23.1.12, and 23.2.2
December 2023	PDF Only	Versions 22.1.12, 22.2.13, 23.1.13, and 23.2.3
February 2024	PDF Only	Versions 22.1.13
March 2024	PDF Only	Versions 22.2.14, 23.1.14, and 23.2.4
April 2024	PDF Only	Version 24.1.1 (R2024a)

Contents

1 User's Guide

2 Fundamentals

3 Run Tasks Using Process Advisor

4 Author Your Process Model

6 Integrate into CI

7 Troubleshooting and Limitations

8 Version History

User's Guide

The support package CI/CD Automation for Simulink® Check™ provides tools to help you integrate your model-based process into a Continuous Integration / Continuous Delivery (CI/CD) system.

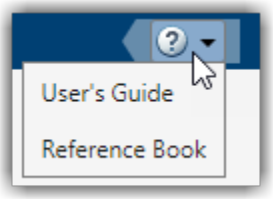
The support package provides:

- A customizable process modeling system that you can use to define your build and verification process
- A build system that can efficiently execute a pipeline in your CI system
- The Process Advisor app for deploying and automating your prequalification process
- Integration with common CI systems, including a pipeline generator to automatically create child pipeline files in CI

You can use the support package to help you set up a model-based design (MBD) pipeline, reduce build time, reduce build failures, debug build failures, and deploy a consistent build and verification process. For an overview of these features, see the chapter "Fundamentals".

This PDF is a User's Guide with general information and examples. For information on the API, artifact types, built-in tasks, and built-in queries, see the Reference Book PDF. You can access the PDFs from:

- https://www.mathworks.com/matlabcentral/fileexchange/115220-ci-cd-automation-for-simulink-check
- The question mark icon in the Process Advisor app



Where to Get Started

If you are a:

- Model developer or test engineer, you might want to start with "Run Tasks Using Process Advisor".
- Process engineer, you might want to start with "Author Your Process Model" and "Control Builds".
- DevOps engineer, you might want to start with "Integrate into CI".

Note For information on the supported releases, features, and compatibility considerations, see the "Version History" at the end of this PDF.

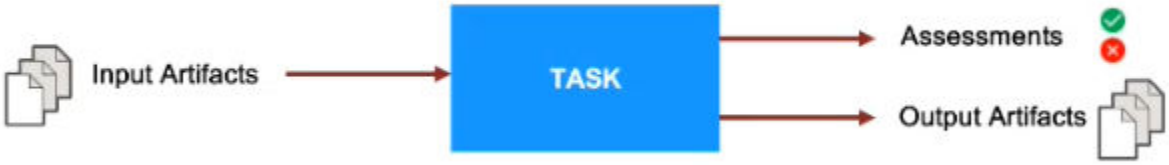
Fundamentals

MBD Pipeline

In a typical CI/CD pipeline, the CI/CD system automatically builds your source code, performs testing, packages deliverables, and deploys the packages to production. With the support package CI/CD Automation for Simulink Check, you can create a pipeline for the steps in your build and verification process, and maintain a repeatable CI/CD process for model-based design. For example, you can create an MBD pipeline that checks modeling standards, runs tests, generates code, and performs a custom task.

You can use the customizable process modeling system to define the steps in your model-based design (MBD) pipeline. You define the steps by using a process model. A *process model* is a MATLAB® script that specifies the tasks in the CI/CD process, dependencies between the tasks, and artifacts that you associate with each task.

A *task* is a single step in your process. Tasks can accept your project artifacts as inputs, perform actions, generate pass, fail, or warning assessments, and return project artifacts as outputs.



The support package contains built-in tasks for several common steps, including:

- Creating Simulink web views for your models with Simulink Report Generator™
- Checking modeling standards with the Model Advisor
- Comparing models to ancestors and generating a comparison report
- Running tests with Simulink Test™
- Detecting design errors with Simulink Design Verifier™
- Generating a System Design Description (SDD) report with Simulink Report Generator
- Generating code with Embedded Coder®
- Checking coding standards with Polyspace® Bug Finder™ and Polyspace Code Prover™
- Inspecting code with Simulink Code Inspector™
- Generating a consolidated test results report and a merged coverage report with Simulink Test and Simulink Coverage™

Tip You can view the source code for the built-in tasks. After installing the support package, the built-in task source code is available in the support package folder. In the MATLAB Command Window, enter:

```
cd(fullfile(matlabshared.supportpkg.getSupportPackageRoot,...
"toolbox","padv","build_service","ml","+padv","+builtin","+task"))
```

This command changes the current working folder to the directory that contains the built-in task source code.

The support package contains a default process model for an MBD pipeline, but you can also customize the default process model to fit your development workflow goals. For example, your process model might include the built-in tasks for checking modeling standards, running tests, and generating code before performing a custom task. You can customize the process model to add or remove any tasks in the MBD pipeline. You can also reconfigure the tasks in your process model to change what action a task performs or how a task performs the action.

For more information on the process modeling system, see the chapter "Author Your Process Model". For information on the built-in tasks, see the chapter "Built-In Task Library" in the Reference Book PDF.

Build System

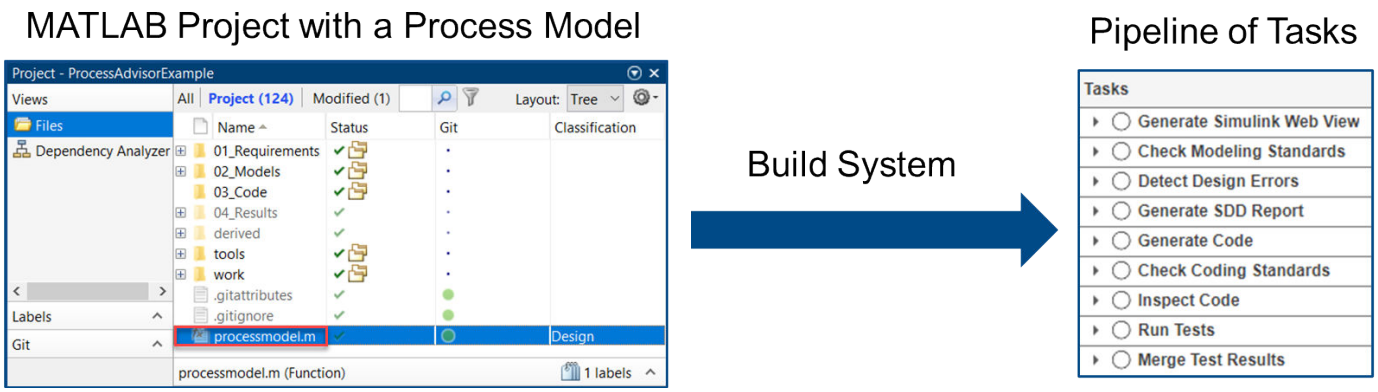
The support package CI/CD Automation for Simulink Check provides a build system that you can use to orchestrate and automate the steps in your MBD pipeline. The *build system* is software that can orchestrate tasks, efficiently execute tasks in the pipeline, and perform other actions related to the pipeline.

The build system needs:

1. A project to analyze
2. A process model in the project that defines the tasks in the pipeline

If the project does not contain a process model, the build system copies the default process model into the project and uses the default process model to define a default MBD pipeline.

When you call the build system, the build system loads the process model, analyzes the project, and orchestrates the creation of a pipeline of tasks.



To run the tasks in the pipeline, you can call the build system using one of these approaches:

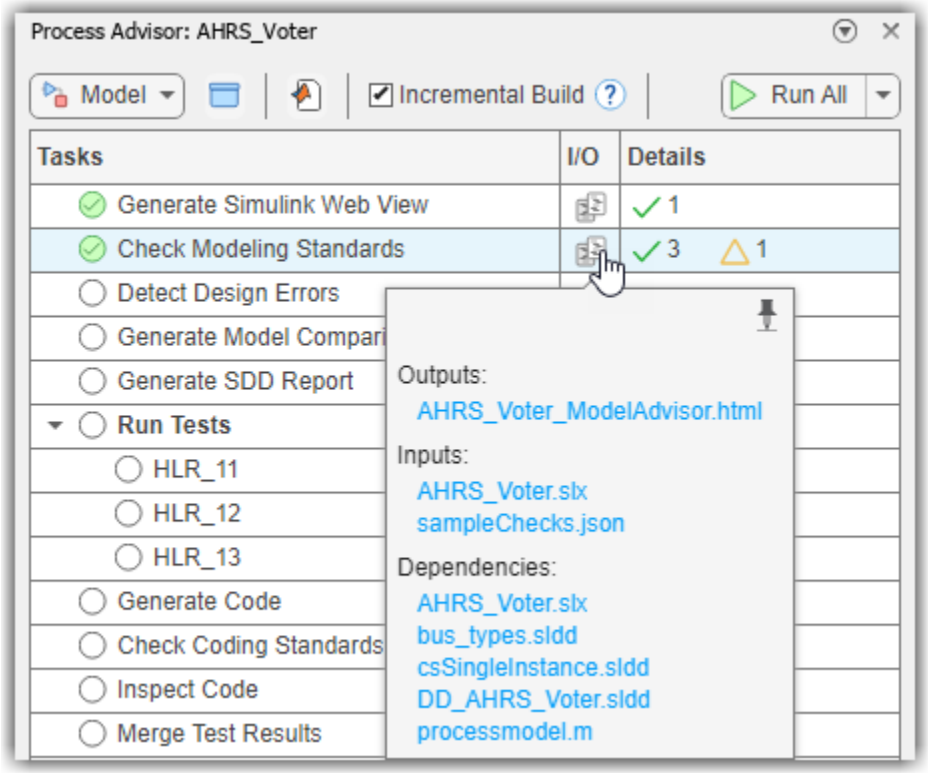
- In a CI environment by using the build system API. The build system API includes a function `runprocess` that you can use to run the tasks in a pipeline.
- Locally on your machine by using either the build system API or the Process Advisor app. Process Advisor is a user interface that can call the build system and has run buttons that you can use to run the tasks in a pipeline. If there is a failure in the CI environment, you can reproduce the issue locally by using Process Advisor on your local machine.

The build system supports incremental builds. If you change an artifact in your project, the build system can detect the change and automatically determine which of the tasks in your MBD pipeline now have outdated results. In your next build, you can instruct the build system to run only the tasks with outdated results. By identifying the tasks with outdated results, the build system can help you reduce build time by reducing the number of tasks you need to re-run after making changes to your project artifacts.

Note There are limitations to the types of changes that the support package can detect. For more information, see the "Limitations on Incremental Build" section.

Process Advisor

A prequalification process can help you prevent build and test failures from occurring in your CI/CD system. Use the Process Advisor desktop app to deploy and automate your prequalification process. You can use the app to run the tasks in your MBD pipeline and to prequalify your changes on your machine before submitting to source control. Process Advisor is a user interface that runs your tasks locally for prequalification. You can run the tasks in your MBD pipeline and to check your progress towards completing tasks in your prequalification pipeline.



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 complete a task but then update your model, the Process Advisor automatically invalidates the task completion and marks the task results as outdated.

Note There are limitations to the types of changes that Process Advisor can detect. For more information, see the "Limitations on Incremental Build".

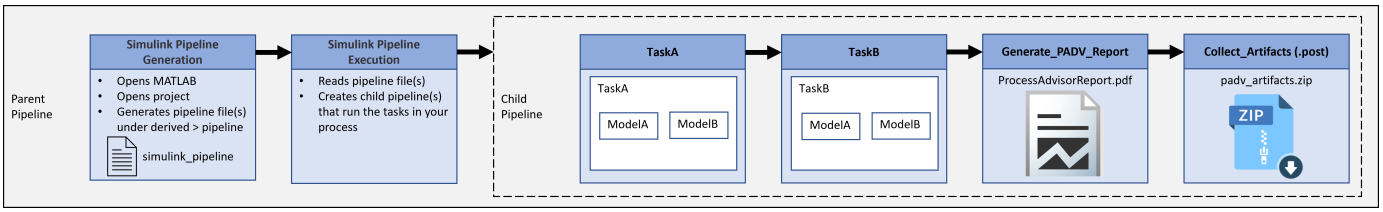
For information on Process Advisor, see "Run Tasks Using Process Advisor".

CI/CD System Integration

You can use the support package CI/CD Automation for Simulink Check to integrate your model-based design process into common CI/CD systems.

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 support package provides a pipeline generator function (`padv.pipeline.generatePipeline`) and example pipeline configuration files that you can use to automatically generate the CI pipelines for you. After you do the initial setup for the pipeline generator, you no longer 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 automatically generate any necessary pipeline configuration files for you.

For example, if your process model contains two tasks, `TaskA` and `TaskB`, the pipeline generator can automatically create a child pipeline that runs the tasks, generates a report, and collects the output artifacts from the CI jobs.



The pipeline generator supports these CI platforms:

- GitHub® — For instructions, see "Integrate into GitHub".
- GitLab® — For instructions, see "Integrate into GitLab".
- Jenkins® — For instructions, see "Integrate into Jenkins".

For information on how to integrate the support package into other CI platforms, see "Integrate into Other CI Platforms".

The support package also contains an example `Dockerfile` for creating a Docker® container to run MATLAB with the support package and other MathWorks® products. For information, see "Create Docker Container for Support Package".

Run Tasks Using Process Advisor

This chapter describes how to use the Process Advisor app to run tasks and prequalify your changes:

Prequalify Changes Before Submitting to Source Control

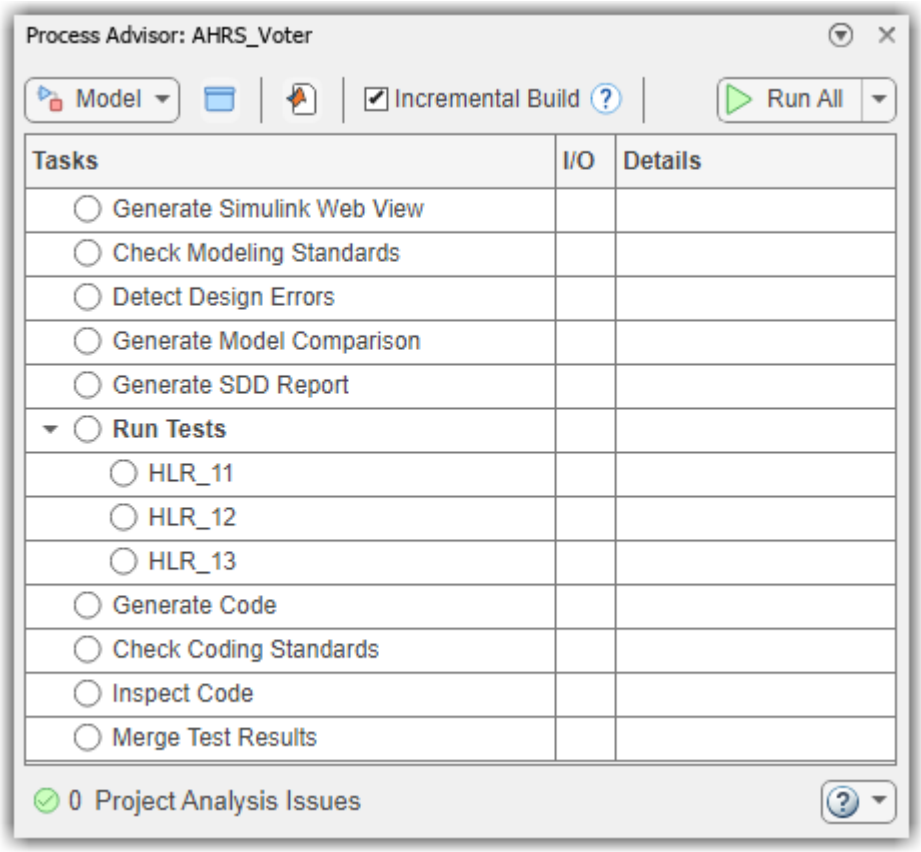
This example shows how to run tasks and review task results using the Process Advisor app. You can use Process Advisor as part of your prequalification process, to run specific tasks before you submit changes to source control. A prequalification process can help you prevent build and test failures from occurring in your continuous integration (CI) system.

1 Process Advisor runs on projects. For this example, open the Process Advisor example project. In the MATLAB Command Window, enter:

```
processAdvisorExampleStart
```

This command creates a copy of the Process Advisor example project and automatically opens Process Advisor on the model `AHRS_Voter`.

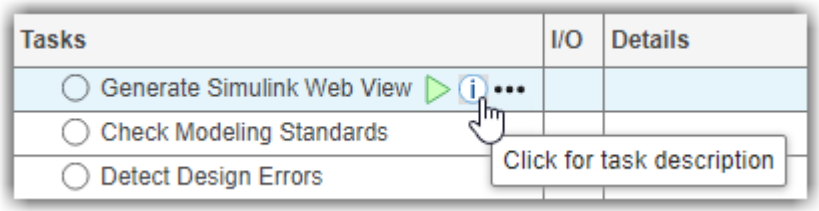
Process Advisor opens in a pane to the left of the Simulink canvas. Process Advisor loads the process model, analyzes the project, and creates a pipeline of tasks. The process model is a file that specifies the tasks in the process and the dependencies between the tasks. The **Tasks** column shows the pipeline of tasks associated with the current model. The tasks appears in the order that the build system will run them. For more information about the process model and tasks, see "About the Process Model".



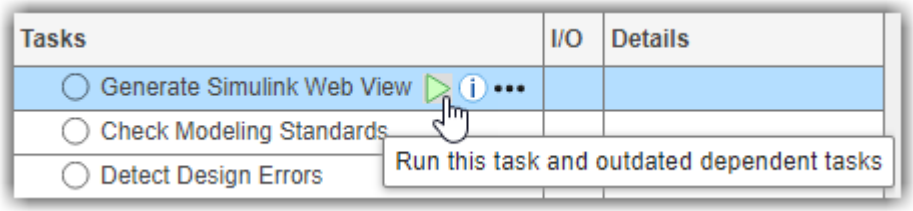
Note If you already have your own project, you can open Process Advisor by clicking **Process Advisor** in the **Project** tab.

If artifact tracing is not already enabled for your project, Process Advisor opens the Enable Artifact Tracing dialog. Click **Enable and Continue** to allow Process Advisor to analyze the artifacts and relationships in your project.

2. To view information about a task, point to the task in the **Tasks** column and click on the information icon. When you click on the information icon, you can view the task description.

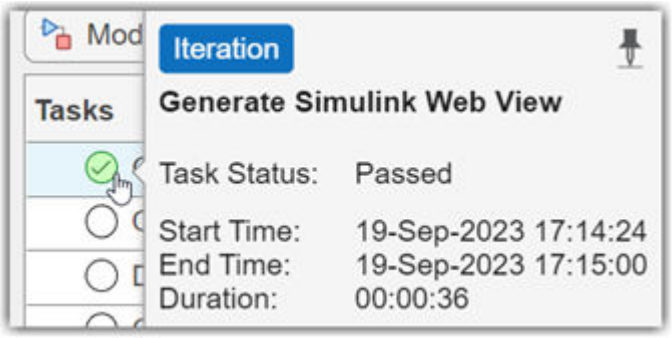


3. You can run an individual task by pointing to the task and clicking the run button. Point to the **Generate Simulink Web View** task and click the run button.

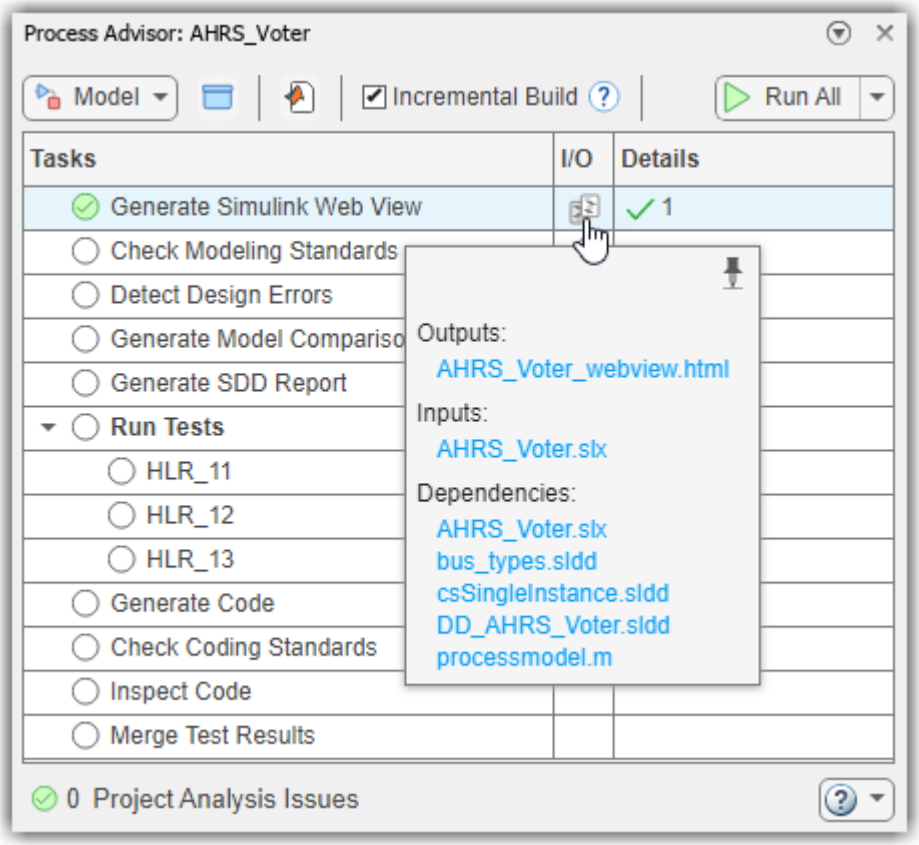


The **Generate Simulink Web View** task runs on the current model. Process Advisor logs task activity in the MATLAB Command Window.

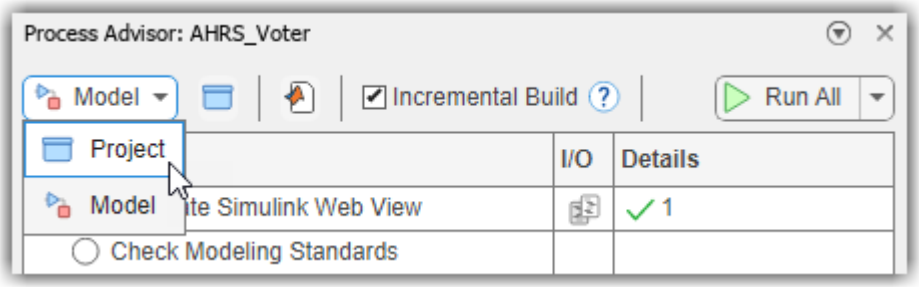
When the task runs successfully, the status in the **Tasks** column shows a green circle with a check mark. When you point to the status icon, you can view details about the status, including the task status and how long the task took to run.



If you point to the file icon in the **I/O** column, the pop-up shows hyperlinks to the outputs from the task, and any inputs and dependencies for the task. In the **Details** column, you can see that the task successfully generated one Simulink web view.



4 In the top-left corner of the Process Advisor pane, switch the filter from **Model** to **Project**.

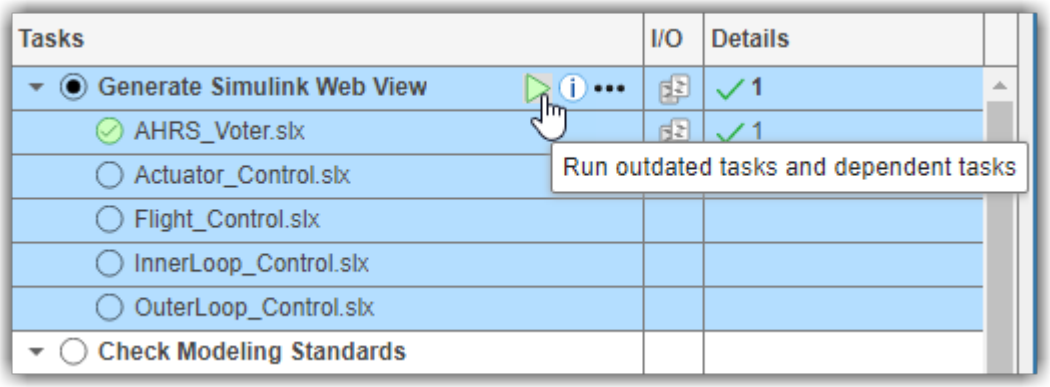


When you set the filter to **Project**, the Process Advisor pane shows the tasks associated with the project. By default, the **Generate Simulink Web View** task is configured to run once on each model in the project. The Process Advisor uses a query to find each of the models in the project and shows the names of the models as individual task iterations below the task title. The task status for **Generate Simulink Web View** shows the multiple statuses icon ◉ because the task passed on the `AHRS_Voter` model and was not run on the other models. For more information on icons, see "Icon Overview".

Note You can click on an artifact name in the **Tasks** column to open the artifact.

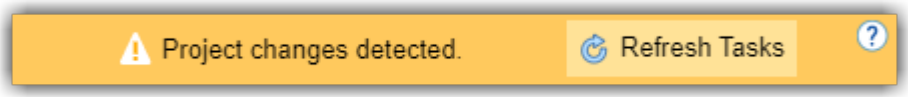
To open a tool associated with the task, point to the task iteration, click the ellipsis (**...**) and then **Open *`Tool Name`***. For example, when you point to the **Generate Simulink Web View** task and click the ellipsis (**...**), you have the option to **Open Web View Options**.

5 Point to **Generate Simulink Web View** and click the run button ▷ to run the task for each model in the project.



6 After the **Generate Simulink Web View** task runs, make a change to the `AHRS_Voter` and re-save the model. For this example, you can click and drag the Model Info block to a different part of the Simulink canvas and re-save the model.

Process Advisor detects the change to the model and shows a warning banner.

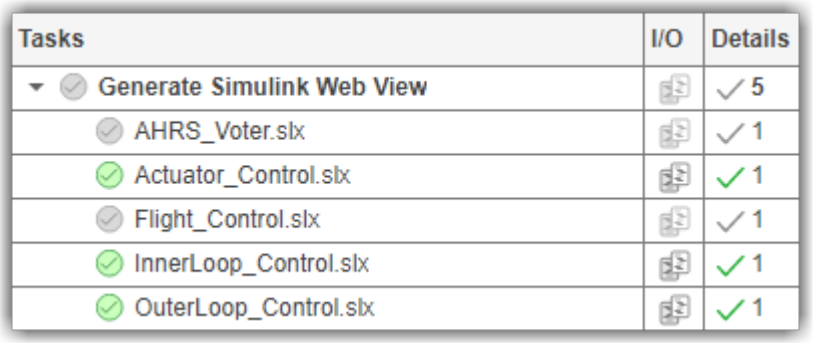


Note There are limitations to the types of changes that the Process Advisor can detect. For more information, see the "Limitations on Incremental Build" section.

Note that sometimes the warning banner might appear while you are running tasks or after you have finished running tasks, depending on when file system events reach MATLAB.

7 Click the **Refresh Tasks** button on the warning banner to refresh the information shown in Process Advisor to reflect the impact of your change on the task statuses.

Process Advisor automatically identified that the **Generate Simulink Web View** task results are outdated for both **AHRS_Voter.slx** and **Flight_Control.slx**. When a task previously passed but now has outdated results, the task status in the **Tasks** column shows the **Passed (Outdated)** icon .

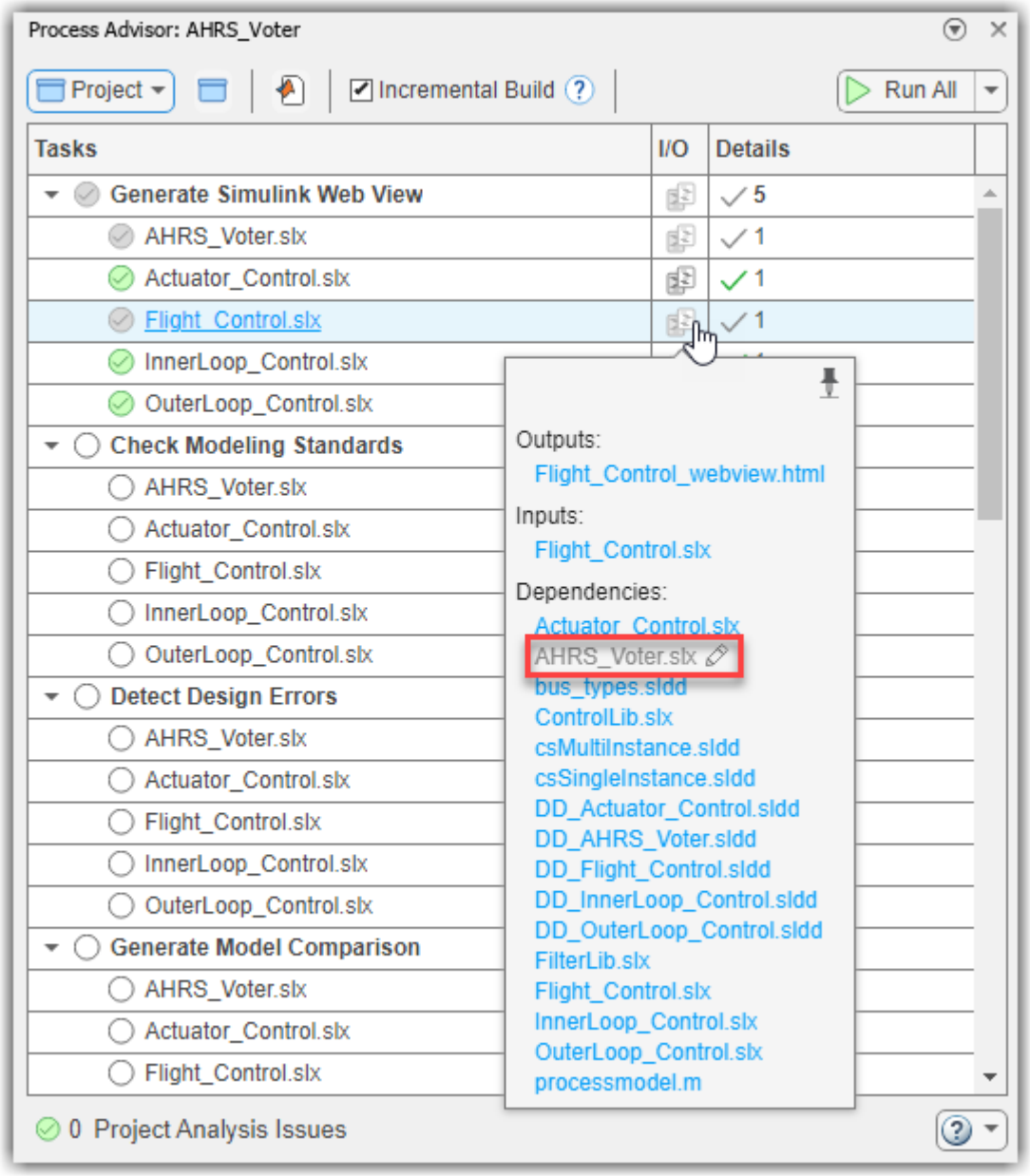


The task results for **AHRS_Voter.slx** are outdated because you modified the model and directly invalidated the task results. The task results for **Flight_Control.slx** are outdated because the `AHRS_Voter` model now has outdated results and `Flight_Control` references the `AHRS_Voter`.

If you point to the file icon in the **I/O** column, the pop-up shows why the task results are stale. The outdated file icon appears next to files that changed and caused the task results to become outdated. In this example, `Flight_Control.slx` depends on the model `AHRS_Voter.slx` and `AHRS_Voter.slx` changed since the last time **Generate Simulink Web View** ran on `Flight_Control.slx`.

Note The build system and Process Advisor app use incremental builds to only rerun tasks with outdated results, skipping tasks that are up-to-date. If you do not want the build system to mark tasks as up-to-date or outdated, click **Settings** and clear the **Incremental Build** setting.

If you want to control whether specific task inputs or outputs impact the task status, see "Turn Off Change Tracking for Input Artifacts" and "Turn Off Change Tracking for Task Outputs".



8 Re-run the **Generate Simulink Web View** task to get updated task results. Point to the **Generate Simulink Web View** task and click the run button ▷.

The build system automatically runs an incremental build that runs only the outdated tasks and skips any tasks that already have up-to-date results.

In the column **Results**, 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.

Process Advisor aggregates the results of each task. For this example, the **Generate Simulink Web View** task successfully created five web views, so the column **Results** shows a value of **5** next to the green check mark for the task.

The log in the MATLAB Command Window shows the build results from running the task, including the number of task iterations that the build system was able to skip because the results were already up-to-date.

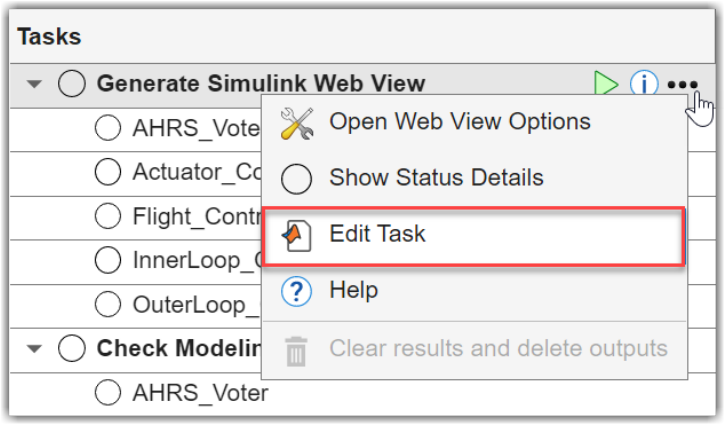
```
#### Build Status:              Pass
#### Number of tasks:           5
#### Number of tasks executed:  2
#### Number of tasks skipped:   3
```

9 Generate a PDF report with the current task results. Create a `padv.ProcessAdvisorReportGenerator` object and call `generateReport` on the object. In the MATLAB Command Window, enter:

```
rptObj = padv.ProcessAdvisorReportGenerator; % create a report object
generateReport(rptObj) % generate a report
```

The report generates in your current working folder. The report summarizes the task statuses, task results, and other information about the task execution. For more information, see the "Generate Build Report" section of this PDF.

10 To view the source code for a task, point to the task, click the ellipsis (**...**), and then click **Edit Task**.



If you want to change the behavior of a built-in task, you can reconfigure the task. For information, see "Author Your Process Model". You can also create and reconfigure multiple instances of a built-in task and create new custom tasks. For information, see "Create Multiple Instances of Tasks" and "Create Custom Task".

When you use the Process Advisor app to prequalify changes before submitting to source control, you can click the **Run All** button to run each of the tasks in your process and confirm that each of your tasks passes. The build system automatically skips tasks that already have up-to-date results and only runs tasks that have outdated results. Process Advisor allows you to confirm that your changes do not cause issues with your existing functionality and helps you to prevent failures in CI.

Note Process Advisor creates a `derived` folder that contains information about your project and task results. Do not add the `derived` folder to your project or to your source control system. The `derived` folder contains derived results that should not be under source control.

Explore Other Options

Use this table to find information based on your goals.

Goal	Related Information
Learn more about the Process Advisor app.	"Quick Reference for Process Advisor App"
Customize the pipeline of tasks by reconfiguring the built-in tasks, removing tasks, and adding custom tasks.	"Author Your Process Model"
Customize how tasks display artifact names in Process Advisor	"Hide File Extension in Process Advisor"
Integrate into a continuous integration (CI) system.	"Integrate into CI"
Debug failures seen in CI.	"Locally Reproduce Issues Found in CI"

Locally Reproduce Issues Found in CI

After you run a pipeline in your CI system, you can 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.

To copy CI results onto your machine:

1. Get the latest changes by cloning a copy of the project onto your local machine. For information, see https://www.mathworks.com/help/simulink/ug/clone-git-repository.html.
2. Close your local copy of the project.
3. In your CI system, open the job that you want to inspect locally and download the artifacts that the job generated. 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.

   For example, in GitLab, you can use either the GitLab UI or API to download job artifacts: https://docs.gitlab.com/ee/ci/pipelines/job_artifacts.html#download-job-artifacts

   Job artifacts typically download as a ZIP file.
4. Extract the files from the ZIP file and copy the artifacts into your local copy of the project. The copied artifacts do not need to be added to the MATLAB path or project path.
5. Open your local copy of the project in MATLAB.
6. Open the Process Advisor app. If there is a warning banner, click **Refresh Tasks**.

After you refresh the tasks, you can:

- 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 any open issues before submitting to source control

Note If you use a parallel pipeline architecture like `IndependentModelPipeline` in releases older than R2023b Update 5, 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.

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".

Quick Reference for Process Advisor App

Process Advisor

Automate your development workflow and prequalify changes before submitting to source control

Description

Use the Process Advisor app to create, deploy, and automate a consistent prequalification process for Model-Based Design (MBD). The app includes built-in tasks for performing common MBD tasks like checking modeling standards with the Model Advisor app, running tests with Simulink Test, generating code with Embedded Coder, and inspecting code with Simulink Code Inspector. You can use the customizable process modeling system to define the steps in your process and use the app to run each of the steps. As you edit and save the artifacts in your project, the app tracks changes and automatically identifies tasks and task iterations that have outdated results. The Process Advisor app runs your tasks locally for prequalification. The tasks run on the machine that is running MATLAB and does not use an external CI system.

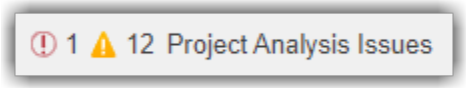
To run tasks:

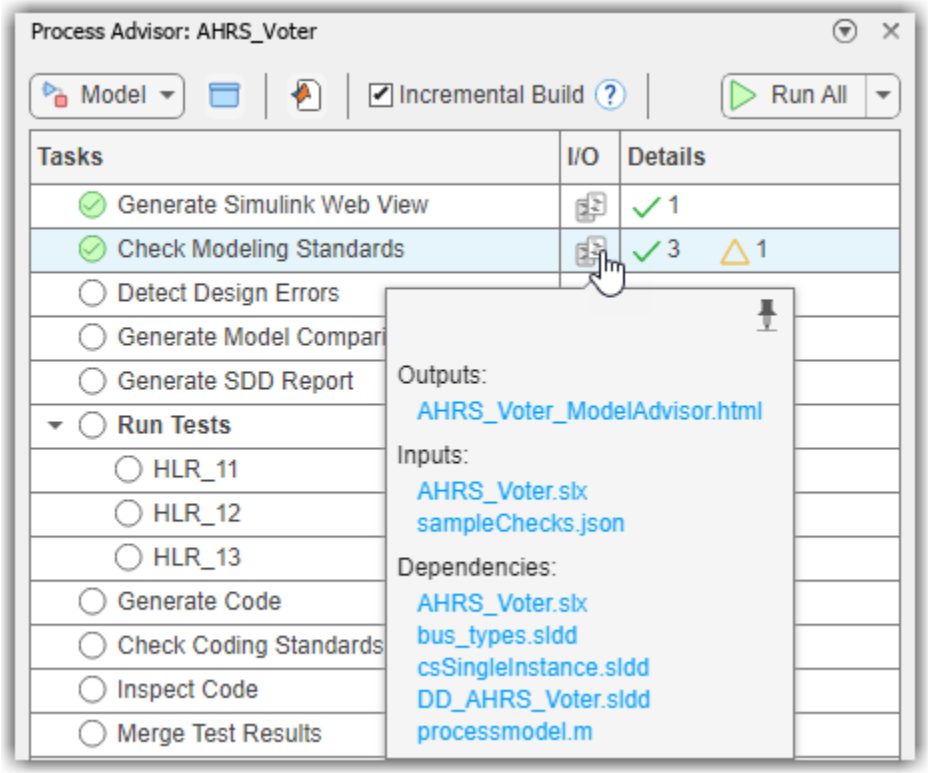
- Point to a task in the **Tasks** column and click the run button ▷ to run that task and any dependent tasks.
- Click **Run All** to run each of the tasks shown in the **Tasks** column.
- Click **Run All** > **Force Run All** to force the build system to run each task, even if the tasks already have up-to-date results.
- Click **Run All** > **Clean All** to clear the task results and delete task outputs for each of the tasks.
- Click **Run All** > **Refresh All** to manually refresh the list of tasks that appears in the **Tasks** column.

When the Process Advisor app runs tasks, a **Stop** button appears in the top-right corner. You can click the **Stop** button to stop the queued tasks from running next.

To edit the process model, click the **Edit process model** icon  . If you have a P-coded process model file, you must delete the `processmodel.p` file before you can edit the process model using Process Advisor.

After Process Advisor analyzes the project, the **Project Analysis Issues** pane shows any errors or warnings that were generated during artifact analysis. For more information, see the "Troubleshooting and Limitations" section.

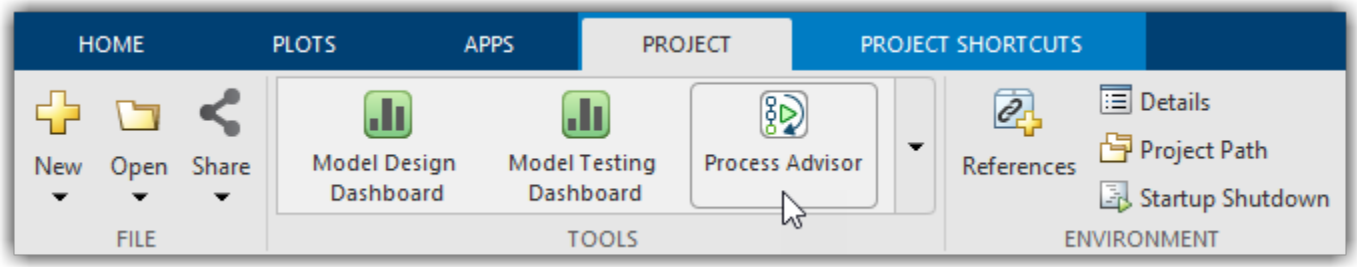




Open the Process Advisor App

- For a Simulink model:
  - On the **Apps** tab, click **Process Advisor**.
  - Or, in the Command Window, enter:

    ```
    processadvisor(modelName)
    ```
- For a project:
  - On the **Project** tab, in the **Tools** section, click **Process Advisor**.



  - Or, in the Command Window, enter:

    ```
    processAdvisorWindow
    ```

Examples

Open Process Advisor For Model

Open the Process Advisor app for a Simulink model in a project.

Create and open a working copy of the Process Advisor example project. MATLAB copies the files to an example folder so that you can edit them.

```
processAdvisorExampleStart
```

The project contains the model `OuterLoop_Control.slx`.

Open the Process Advisor app for the model `OuterLoop_Control.slx`.

```
processadvisor("OuterLoop_Control")
```

Open Process Advisor For Project

Open the Process Advisor for a project and view the pipeline of tasks.

Create and open a working copy of an example project. MATLAB copies the files to an example folder so that you can edit them.

```
proj = Simulink.createFromTemplate("code_generation_example.sltx",...
Name="New Project");
```

Open the Process Advisor for the project.

```
processAdvisorWindow
```

The **Tasks** column shows the pipeline of tasks generated from the process model.

Click **Edit**  to open the `processmodel.m` file that defines the process.

Programmatic Use

Note that you need to load a project before you open the Process Advisor.

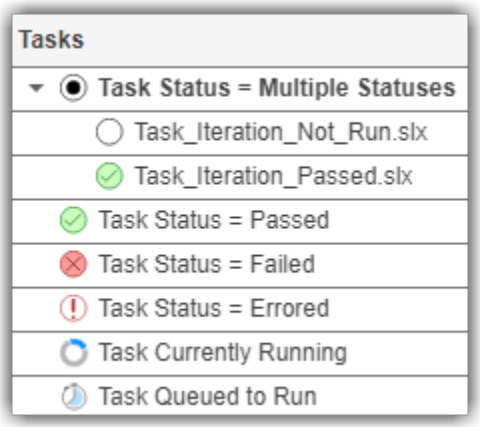
`processadvisor(modelName)` opens the Simulink model, `modelName`, in the current project and opens a Process Advisor pane to the left of the Simulink canvas.

`processAdvisorWindow()` opens the Process Advisor app for the current project. The app opens in a standalone window.

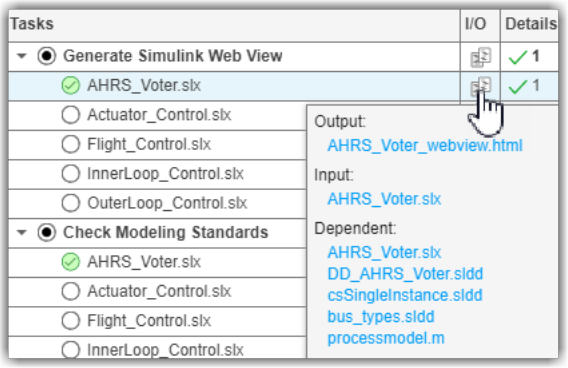
Icon Overview

The Process Advisor app uses the:

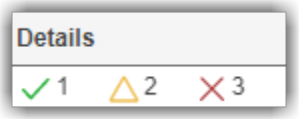
- **Tasks** column to show the statuses for the tasks and task iterations.



- **I/O** column to show the outputs from the tasks and task iterations.



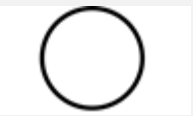
- **Details** column to show detailed results for tasks and task iterations that specify result values.



Tasks Column

The status for the task or task iteration is shown on the left side of the **Tasks** column.

Statuses in the Tasks Column

Icon	Status of the Task or Task Iteration	Icon When Results Outdated	Icon When Incremental Builds Turned Off
	Not run.	Not applicable.	Uses same icon.
	Currently running.	Not applicable.	Uses same icon.
	Queued to run during the current build.	Not applicable.	Uses same icon.
	Passed.		
	Failed.		
	Generated an error.		
	Multiple statuses for different iterations of a task.		Uses same icon.

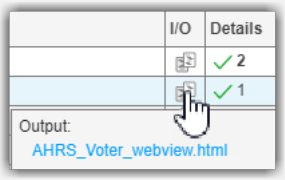
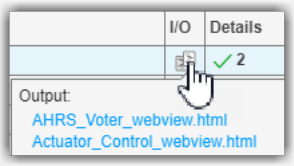
For more information on the task statuses, see the documentation for the `Status` property of the `padv.TaskResult` class in the Reference Book PDF.

Note Tasks that generated an error do not rerun automatically. To rerun an errored task, point to the task and click the run button or use `runprocess` with `RerunErroredTasks` as `true`.

I/O Column

The Process Advisor app shows the outputs from a task or task iteration when you point to the icon in the **I/O** column.

Outputs in the I/O Column

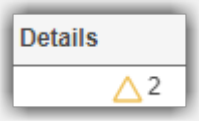
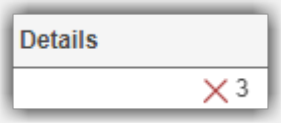
Icon	Description	Icon When Outdated
	The task or task iteration output a single artifact. 	
	The task or task iteration output multiple artifacts. 	

For more information on the outputs, see the documentation for the `OutputArtifacts` property of the `padv.TaskResult` class in the Reference Book PDF.

Details Column

Detailed results from a task or task iteration are shown in the **Details** column.

Results in the Details Column

Icon	Result Value	Result Value for the Task or Task Iteration	Icon When Outdated
	Pass.	The value to the right of the icon indicates the number of result values that passed. 	
	Warn.	The value to the right of the icon indicates the number of result values that generated a warning. For example, the **Check Modeling Standards** task shows the number of Model Advisor checks that generated a warning. Review the outputs in the **I/O** column and any other results from the task to identify the issue. 	
	Fail.	The value to the right of the icon indicates the number of result values that failed. Review any reports, outputs, or other results from the task. 	

For more information on the detailed results, see the documentation for the `ResultValues` property of the `padv.TaskResult` class in the Reference Book PDF.

Author Your Process Model

This chapter describes how to use the customizable process modeling system to define your build and verification process:

Tip You can access API help from the MATLAB Command Window by using the `help` function.

For example, this code returns help information for the class `padv.Task`:

```
help padv.Task
```

The Reference Book PDF also includes documentation for the API and built-ins.

About the Process Model

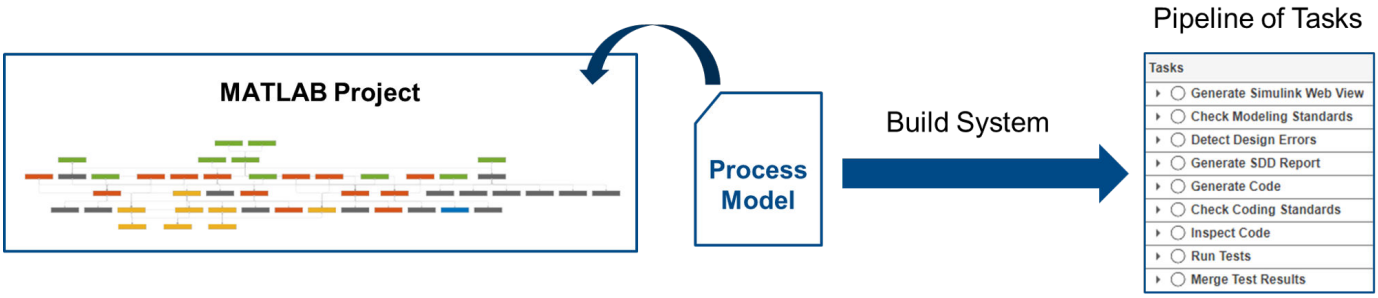
The support package has a customizable process modeling system that you can use to define your process. The support package also has a build system and front-end (Process Advisor app) for managing, deploying, and using your process. You can run the build system and Process Advisor locally on your desktop, and you can run the same build system in your CI environment.

The support package includes a default process model that can create an MBD pipeline. The default process model can create an MBD pipeline that contains several common model-based design tasks. You can modify the default `processmodel.m` file to fit your development process goals or you can create a new process model from an empty template. For more information, see "Modify Default Process Model to Fit Your Process".

Requirements

The Process Advisor app requires you to have:

- Your files in a project.
- A process model file (`processmodel.p` or `processmodel.m`) on the MATLAB path. If possible, place your process model file in the project root folder so changes to the process model file are tracked. If your project does not have a process model and you open the Process Advisor app, the Process Advisor automatically creates a default process model for you at the root of the project.



You define your pipeline of tasks in the process model. The *process model* is a file that specifies the tasks in the process, queries that determine which artifacts to use for each task, artifacts associated with each task, and dependencies between the tasks.

Your file serves as the process model if it meets the following criteria:

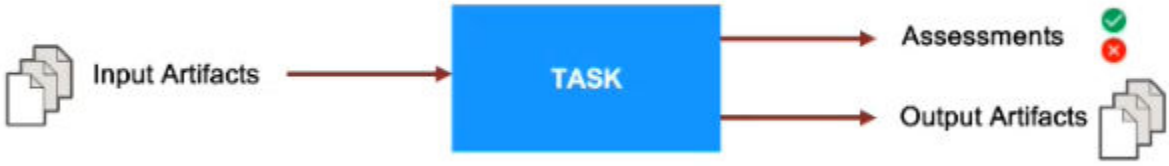
- The filename is `processmodel.p` or `processmodel.m`. If you have both a P-code file and a `.m` file, the P-code file takes precedence over the corresponding `.m` file for execution, even after modifications to the `.m` file.
- The file is in the project root folder.

You do not need to manually run the process model. The process model only defines the tasks that you want to include in your pipeline. When you run tasks by using the Process Advisor app or the build system API, the build system automatically loads the process model to create your pipeline of tasks.

Tasks and Queries

The process modeling system allows you to manage your process by using:

- Tasks — Individual steps in your process. Tasks can accept your project artifacts as inputs, perform actions, generate pass, fail, or warning assessments, and return project artifacts as outputs. Your process is a collection of steps that you want to perform on a project. There are built-in tasks for common tasks like running Model Advisor checks, generating code, and running tests, but you can also reconfigure the built-in tasks or create new custom tasks. For more information on the built-ins, see the "Built-In Task Library" in the Reference PDF. For information on custom tasks, see "Create Custom Task".



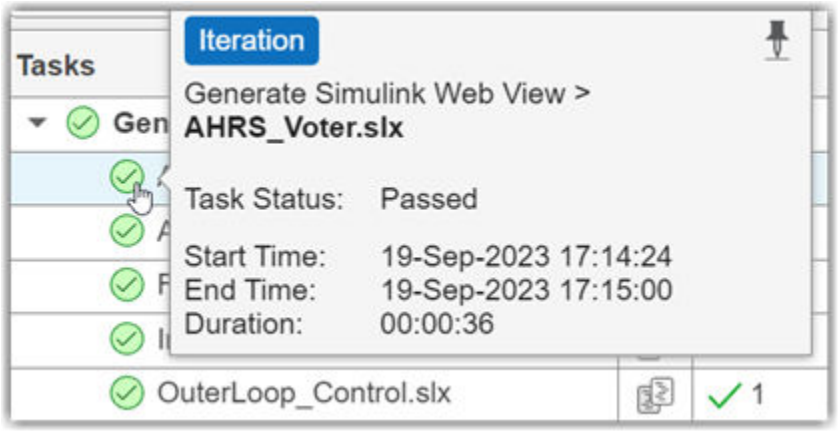
- Queries — Find artifacts in your project automatically, without needing to manually update a static list of files. Queries can automatically analyze all artifacts under your project root folder, even if you did not add the files to the project. You can use queries to find artifacts based on the artifact type, project label, file path, and other properties. There are built-in queries for finding artifacts based on specific search criteria, finding top models, and finding the artifact that a task performs an action on, but you can also create your own custom queries. For more information, see "Change How Often Tasks Run", "Add Inputs to Tasks", and "Create Custom Query".

When you add a task to your process model, you can use queries to specify:

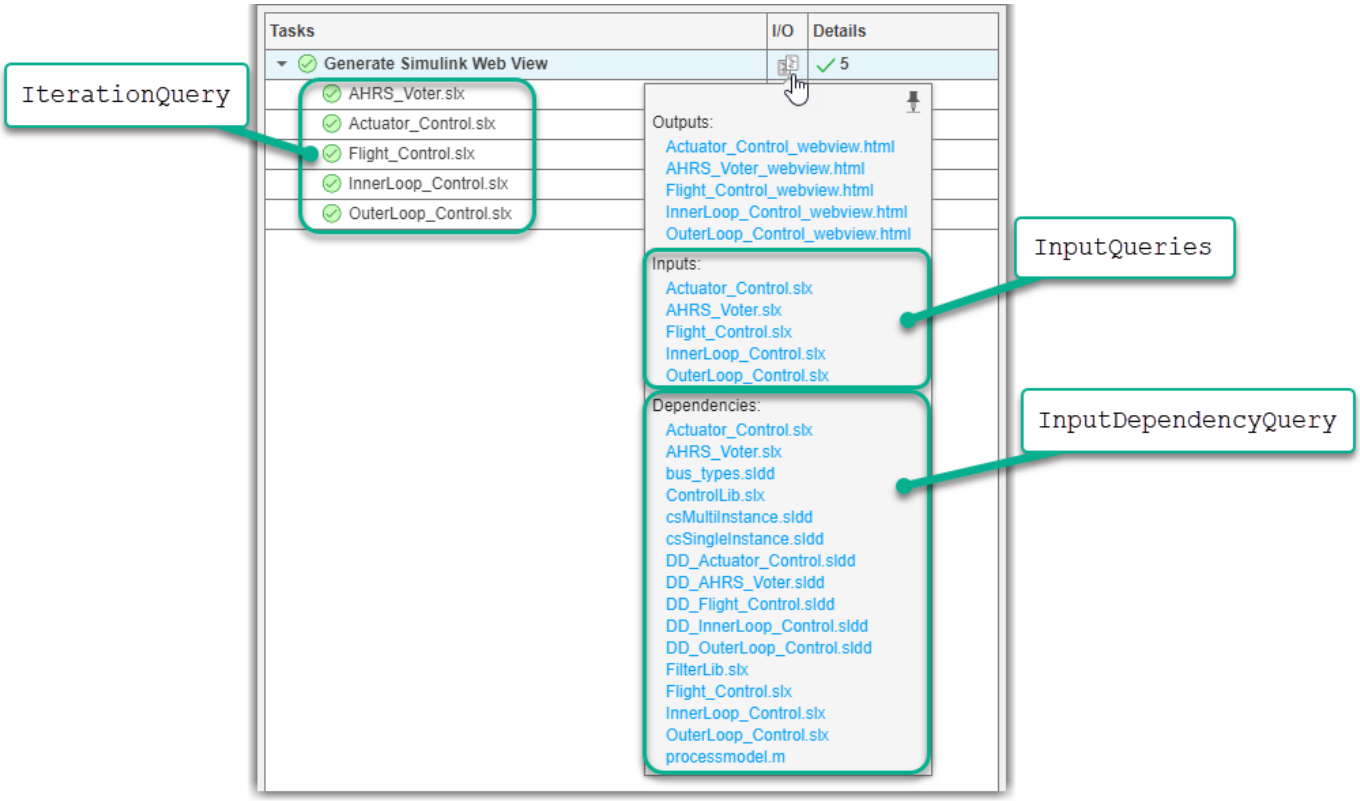
- How often the task runs (defined by the `IterationQuery`)
- Additional inputs to the task (defined by the `InputQueries`)

For each task in the process, the build system runs the `IterationQuery` to determine which artifacts to run the task for. Most built-in tasks use the iteration query `IterationQuery = "padv.builtin.query.FindModels"` to run the task once for each model in the project. The build system then creates a task iteration, runs any additional queries the task needs, runs the task, and saves the task results. The task iteration is the pairing of the task to a specific artifact, for example running the **Generate Simulink Web View** task for the model `AHRS_Voter.slx`. The task iterations appear below the task title in the **Tasks** column in Process Advisor. If the iteration query does not return any results, the task no longer appears in Process Advisor.

If you point to the status, you can see the task and task iteration information. For example, **Generate Simulink Web View** represents an abstract **Task** and **AHRS_Voter.slx** represents a task **Iteration**.



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 any additional dependencies that can affect whether task results are up-to-date. The task inputs appear under **Inputs** and the additional dependencies appear under **Depenendencies** in the **I/O** column in Process Advisor.



If you see artifacts in the **Tasks** column that you want Process Advisor to ignore, you can change the behavior of the task to filter out those artifacts. For information, see "Change How Often Tasks Run". If you want to add inputs to a task, see "Add Inputs to Tasks".

Modify Default Process Model to Fit Your Process

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.

Create Process for Project

1 Open a project. You can use your own project or you can use a Process Advisor example project:

- For the default example project, enter:

  ```
  processAdvisorExampleStart
  ```

- For an example project designed for parallel CI jobs, enter:

  ```
  processAdvisorParallelExampleStart
  ```

2 Open Process Advisor on the project. In the **Project** tab, click **Process Advisor** or enter:

```
processAdvisorWindow
```

If your project does not have a process model, Process Advisor automatically creates a process model file, `processmodel.m`, at the root of the project. The `processmodel.m` file serves as the definition for your process. You do not need to manually run the `processmodel.m` file. Process Advisor automatically reads the process model and uses the file to determine which tasks to run, how the tasks perform their actions, and in which order the tasks need to run. The tasks defined in the process model appear in the **Tasks** column in Process Advisor and appear in the order that they run.

Note Alternatively, you can programmatically create a new process model by using the `createprocess` function. For example:

```
createprocess(Template="default",Overwrite=true)
```

For a process model designed for parallel CI jobs:

```
createprocess(Template="parallel",Overwrite=true)
```

Inspect Process

Inspect the process model. In the Process Advisor window, click the **Edit** button  .

Process Advisor opens the process model at the root of the project. 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. But you can customize the process model to reconfigure the built-in tasks, add custom tasks, or remove tasks.

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

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

    arguments
        pm padv.ProcessModel
    end

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

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


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %% Define Variables
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Set default root directory for task results
    pm.DefaultOutputDirectory = fullfile('$PROJECTROOT$', 'PA_Results');
    defaultResultPath = fullfile( ...
        '$DEFAULTOUTPUTDIR$','$ITERATIONARTIFACT$');

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

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

    %% Run Design Error Detection (DED) on a model
    if includeDesignErrorDetectionTask
        dedTask = pm.addTask(padv.builtin.task.DetectDesignErrors()); %#ok<*UNRCH>
        dedTask.ReportFilePath = fullfile( ...
            defaultResultPath, 'design_error_detections','$ITERATIONARTIFACT$_DED');
    end
```

```
100
101     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
102     %% Set Task relationships
103     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
104
105     %% Set Task Dependencies
106     if includeGenerateCodeTask && includeCodeInspection
107         slciTask.dependsOn(codegenTask);
108     end
109     if includeGenerateCodeTask && includeAnalyzeModelCode
110         psTask.dependsOn(codegenTask);
111     end
112     if includeTestsPerTestCaseTask && includeMergeTestResultsTask
113         mergeTestTask.dependsOn(milTask, "WhenStatus",{'Pass','Fail'});
114     end
115
116     %% Set Task Run-Order
117     if includeModelStandardsTask && includeSimulinkWebViewTask
118         maTask.runsAfter(slwebTask);
119     end
120     if includeDesignErrorDetectionTask && includeModelStandardsTask
121         dedTask.runsAfter(maTask); %#ok<*NODEF>
122     end
123     if includeSDDTask && includeModelStandardsTask
124         sddTask.runsAfter(maTask);
125     end
126     if includeTestsPerTestCaseTask && includeModelStandardsTask
127         milTask.runsAfter(maTask);
128     end
129     % Set the code generation task to always run after Model Standards,
130     % System Design Description and Test tasks
131     if includeGenerateCodeTask && includeModelStandardsTask
132         codegenTask.runsAfter(maTask);
133     end
134     if includeGenerateCodeTask && includeSDDTask
135         codegenTask.runsAfter(sddTask);
136     end
137     if includeGenerateCodeTask && includeTestsPerTestCaseTask
138         codegenTask.runsAfter(milTask);
139     end
140     % Both the Polyspace Bug Finder (PSBF) and the Simulink Code Inspector
141     % (SLCI) tasks depend on the code generation tasks. SLCI task is set to
142     % run after the PSBF task without establishing an execution dependency
143     % by using 'runsAfter'.
144     if includeGenerateCodeTask && includeAnalyzeModelCode ...
145             && includeCodeInspection
146         slciTask.runsAfter(psTask);
147     end
148
```

C

D

Section A — Add or Remove Built-In Tasks

This section of the process model defines which built-in tasks are added to the process:

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

includeModelStandardsTask = true;
includeDesignErrorDetectionTask = false;
includeModelComparisonTask = false;
includeSDDTask = true;
includeSimulinkWebViewTask = true;
includeTestsPerTestCaseTask = true;
includeMergeTestResultsTask = true;
includeGenerateCodeTask = true;
includeAnalyzeModelCode = true && ...
                          ~padv.internal.isMACA64 && ...
                          exist('polyspaceroot','file');
includeProveCodeQuality = true && ...
                          ~isempty(ver('pscodeprover')) || ...
                          ~isempty(ver('pscodeproverserver'));
includeCodeInspection = false;
```

You can update this section to add or remove built-in tasks from your process by setting the variable associated with a task to `true` or `false`.

For example, to add the design error detection task to your process, you can change line 13 in your `processmodel.m` file to specify:

```
includeDesignErrorDetectionTask = true;
```

The following table maps the variables in the process model to the associated built-in task title that appears in Process Advisor.

Variable	Task Title in Process Advisor
`includeModelStandardsTask`	**Check Modeling Standards**
`includeDesignErrorDetectionTask`	**Detect Design Errors**
`includeModelComparisonTask`	**Generate Model Comparison**
`includeSDDTask`	**Generate SDD Report**
`includeSimulinkWebViewTask`	**Generate Simulink Web View**
`includeTestsPerTestCaseTask`	**Run Tests**
`includeMergeTestResultsTask`	**Merge Test Results**
`includeGenerateCodeTask`	**Generate Code**
`includeAnalyzeModelCode`	**Check Coding Standards**
`includeProveCodeQuality`	**Prove Code Quality**
`includeCodeInspection`	**Inspect Code**

For information on the built-in tasks, see "Built-In Task Library" in the Reference Book PDF. In addition to the built-in tasks, you can also add custom tasks to your process model. For information on how to create and use custom tasks, see "Create Custom Task".

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 the following built-in tasks in your process model:

- **Generate SDD Report**
- **Generate Simulink Web View**
- **Generate Model Comparison**

For information, see "Set Up Virtual Display for No-Display Machine".

Section B — Change Behavior of Built-In Tasks

This section of the process model changes the values of built-in task properties to change how the tasks perform their actions:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Register Tasks
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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

...
```

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. The default process model specifies that, for this process, the task should save the Model Advisor report in a subfolder named `model_standards_results`.

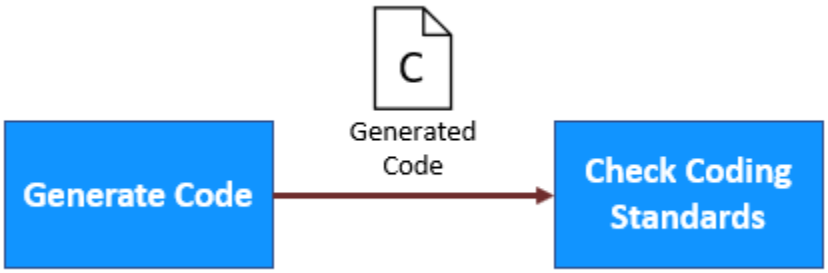
For more information on how to use the properties of built-in tasks to change their behavior, see "Change Task Behavior".

Section C — Specify Dependencies Between Tasks

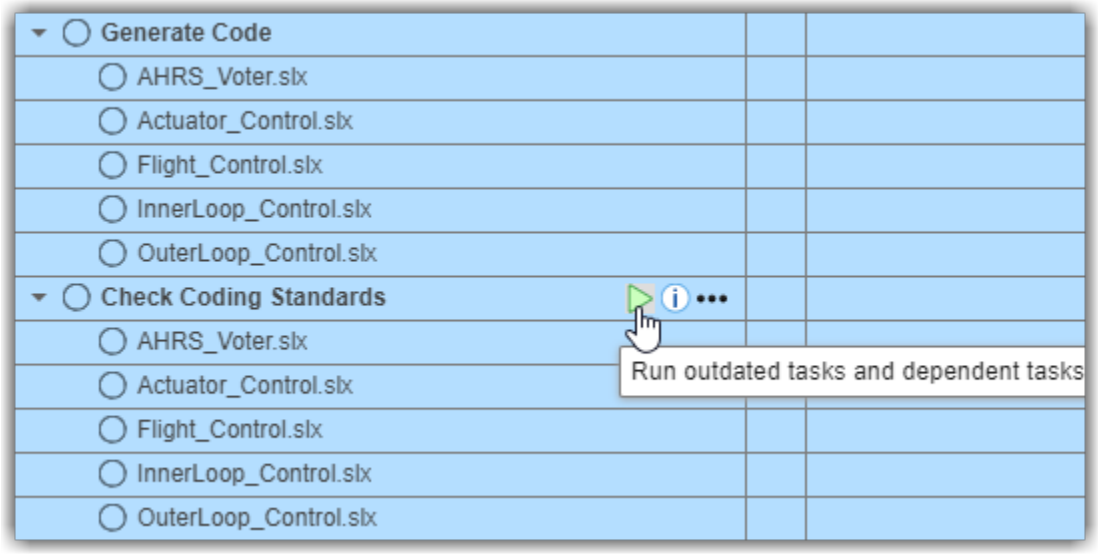
This section of the process model uses the `dependsOn` function to specify which tasks depend on other tasks in order to run successfully:

```
%% Set Task Dependencies
if includeGenerateCodeTask && includeCodeInspection
    slciTask.dependsOn(codegenTask);
end
if includeGenerateCodeTask && includeAnalyzeModelCode
    psTask.dependsOn(codegenTask);
end
if includeTestsPerTestCaseTask && includeMergeTestResultsTask
    mergeTestTask.dependsOn(milTask,"WhenStatus",{'Pass','Fail'});
end
```

For example, you need to generate code before you can use Polyspace to analyze the code. So the default process model specifies that the Polyspace task (`psTask`) depends on the code generation task (`codegenTask`).



If you open Process Advisor and point to the Polyspace task, Process Advisor highlights the dependency between the tasks. If you try to run the Polyspace task, the build system automatically runs the code generation task first.



For more information on task dependencies, see "Specify Dependencies Between Tasks".

Section D — Specify Preferred Task Execution Order

This section of the process model uses the `runsAfter` function to specify a preferred execution order for specific tasks:

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

...
```

These tasks do not need to run in this order to run successfully, but the `runsAfter` function specifies that, if possible, the build system should try to run the tasks in this order.

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

For more information on task ordering, see "Specify Preferred Task Order".

Configure Tasks

Change Task Behavior

You can change the behavior of a built-in task by overriding the values of built-in task properties in the process model.

For example, the built-in task `padv.builtin.task.RunModelStandards` has several properties, like `CheckIDList`, `DisplayResults`, and `ExtensiveAnalysis`.

```
padv.builtin.task.RunModelStandards
```

```
    ans =

      RunModelStandards with properties:

                        CheckIDList: <missing>
                     DisplayResults: "Summary"
                  ExtensiveAnalysis: "on"
                              Force: "on"
                       ParallelMode: "off"
                       ReportFormat: "html"
                       ...
```

The task uses these properties to specify input arguments for the function `ModelAdvisor.run`. The property `CheckIDList` allows you to specify a list of Model Advisor checks that you want the task to run.

By default, the `padv.builtin.task.RunModelStandards` task runs a subset of high-integrity systems checks. But if you specify a new value for the `CheckIDList` property in the process model, the task will run those Model Advisor checks instead:

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

    % Specify which Model Advisor checks to run
    maTask.CheckIDList = {'mathworks.jmaab.db_0032',...
        'mathworks.jmaab.jc_0281'};

end
```

Note This example code shows how to specify a list of Model Advisor checks for the task to run. If you want to use a Model Advisor configuration file instead, you need to provide the configuration file as an input to the task. For information, see "Add Inputs to Tasks".

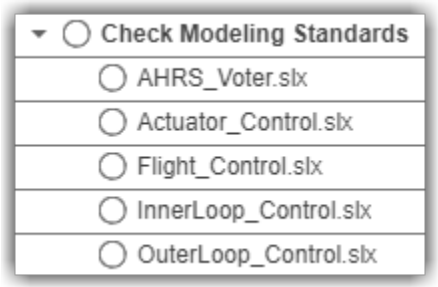
For information on how to reconfigure the **Check Modeling Standards** task to use Model Advisor justification files, see "Check Modeling Standards" in the Reference Book PDF.

For information on the built-in task properties, see the "Built-In Task Library" in the Reference Book PDF or open the source code for the built-in task. For example:

```
open padv.builtin.task.RunModelStandards
```

Change How Often Tasks Run

Most built-in tasks run once for each model in the project. For example, in the Process Advisor example project (`processAdvisorExampleStart`), the task **Check Modeling Standards** runs once for each of these models in the project and the model names appear below the task title in Process Advisor.



However, you can change the `IterationQuery` for a task to specify a different set of artifacts for the task. You must specify the value of `IterationQuery` as either a `padv.Query` object or the name of a `padv.Query`. For each task in the process, the build system runs the iteration query to determine which artifacts to run the task for. By default, the built-in tasks consider the artifacts returned by the iteration query as inputs to the task. Therefore the built-in tasks are able to run on each of the artifacts returned by the iteration query. The support package contains several built-in queries that you can use.

The most commonly used built-in queries are:

- `padv.builtin.query.FindModels` — Find models in the project
- `padv.builtin.query.FindTestCasesForModel` — Find test cases associated with a specific model in the project
- `padv.builtin.query.FindArtifacts` — Finds artifacts in the project that meet the criteria specified in the input arguments

Additionally, some built-in queries have optional arguments that you can use to filter certain artifacts out of the query results.

For information on the built-in queries, see the "Built-In Query Library" in the Reference Book PDF.

Tip You can also access help for the built-in queries from the MATLAB Command Window. For example, this code returns help information for the built-in query `padv.builtin.query.FindArtifacts`:

```
help padv.builtin.query.FindArtifacts
```

Only Run for Specific Models

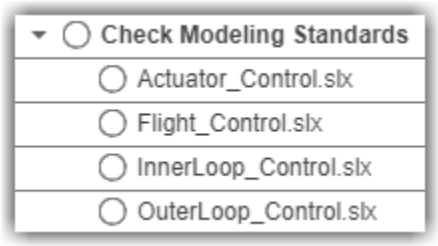
By default, the **Check Modeling Standards** task uses the built-in query `padv.builtin.query.FindModels` as the `IterationQuery`.

But suppose that you only want to run the **Check Modeling Standards** task for models that have `Control` in their file path. In the process model, you can change the `IterationQuery` for the task to:

1. Use the built-in query `padv.builtin.query.FindModels` to find the models in the project
2. Specify the `IncludePath` argument of the query to filter out any models that do not have `Control` in the file path

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

    % Specify which set of artifacts to run for
    maTask.IterationQuery = ...
        padv.builtin.query.FindModels(IncludePath = 'Control')

end
```

In Process Advisor, the model `AHRS_Voter.slx` no longer appears under the task because `AHRS_Voter.slx` does not include `Control` in the path.



Only Run for Specific Test Cases

By default, the **Run Tests** task in the default process model uses the built-in query `padv.builtin.query.FindTestCasesForModel` as the `IterationQuery`. This means that the task runs once for each test case associated with models in the project.

But suppose that you only want to run the task for tests that use a specific project label. In the process model, you can change the `IterationQuery` for the task to:

1. Use the built-in query `padv.builtin.query.FindTestCasesForModel` to find the models in the project
2. Specify the `IncludeLabel` argument of the query to only include test cases that use a specific project label. In this example, the project label is `ModelTest` and the project label category is `TestType`.

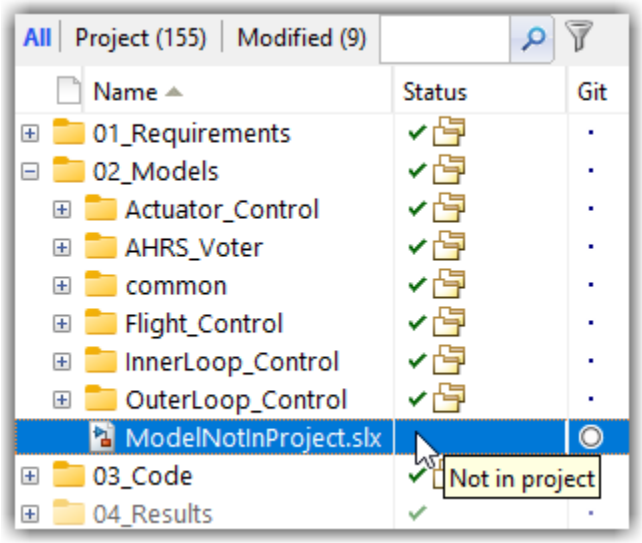
```
%% Running tests on test case to test case basis
if includeTestsPerTestCaseTask
    milTask = pm.addTask(padv.builtin.task.RunTestsPerTestCase());
    % Configure the tests per testcase task
    milTask.OutputDirectory = fullfile( ...
        '$PROJECTROOT$','PA_Results','test_results');

    % Specify which set of artifacts to run for
```

```
    milTask.IterationQuery = ...
        padv.builtin.query.FindTestCasesForModel(IncludeLabel = {'TestType','ModelTest'});

end
```

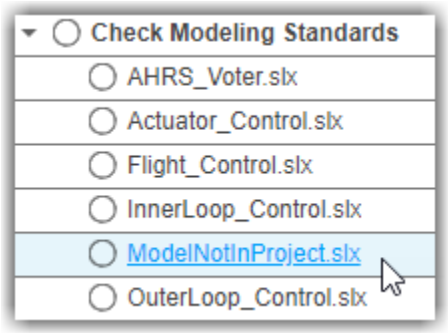
For more information on the built-in queries, see "Built-In Query Library" in the Reference Book PDF. If you need to perform a query that is not already covered by a built-in query, see "Create Custom Query".

Only Run for Artifacts Added to Project

By default, a query can find any artifact under the project root folder, even if you did not add that file to the project. For example, suppose that you have a model that is under your project root folder, but that you did not add to your project.



The model appears in the **Tasks** column in Process Advisor because the built-in queries analyze all artifacts under the project root folder.



If you only want to run a task for models that you added to the project, you can specify the `InProject` argument for the iteration query as `true`. For example, for the **Check Modeling Standards** task, you can update the process model to specify:

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
maTask.IterationQuery = padv.builtin.query.FindModels(...
    InProject = true);
```

Add Inputs to Tasks

By default, the built-in tasks automatically consider the artifacts returned by the `IterationQuery` as input artifacts to the task. But if you want to provide additional inputs to a task, you can add inputs to a task by using the `addInputQueries` function. The `addInputQueries` function adds input queries to the `InputQueries` property of the task. When you run a task, the build system runs the input queries of the task to find the input artifacts that the task can run on. For each input, the build system also runs the `InputDependencyQuery` to find additional input dependencies that can affect whether task results are up-to-date.

Use File as Input to Task

For example, by default, the **Check Modeling Standards** task runs a subset of high-integrity checks. But suppose that you want the task to run the Model Advisor checks specified by the Model Advisor configuration file `sampleChecks.json` instead. You do not need to change any property values, but you do need to add that file as an input to the task. When you provide the file as an input to the task, the task can use the file, recognize changes to the file, and update the task status as needed.

In the process model, you can use the `addInputQueries` function to specify an input query that finds the Model Advisor configuration file. You can use the built-in query `padv.builtin.query.FindFileWithAddress` as an input query to find the Model Advisor configuration file:

- The first argument, `'ma_config_file'`, specifies that the file is a Model Advisor configuration file.
- The second argument specifies the path to the Model Advisor configuration file.

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

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

end
```

For information on how to reconfigure the **Check Modeling Standards** task to use a Model Advisor configuration file or Model Advisor justification files, see "Check Modeling Standards" in the Reference Book PDF.

Note If you provide both a list of check IDs (`CheckIDList`) and a Model Advisor configuration file for the task, the task runs Model Advisor using the Model Advisor configuration file and ignores the list of check IDs.

Use Task Outputs as Task Inputs

Suppose that you want to pass the output of one task as the input to another task. You can use the built-in query `padv.builtin.query.GetOutputsOfDependentTask` to find the outputs of the predecessor task and specify that query as an input query for the task.

For example, the default process model specifies that the **Merge Test Results** task depends on the **Run Tests** task:

```
if includeTestsPerTestCaseTask && includeMergeTestResultsTask
    mergeTestTask.dependsOn(milTask, "WhenStatus",{'Pass','Fail'});
end
```

If you open the source code for the **Merge Test Results** task, you can see that the task uses the built-in query `padv.builtin.query.GetOutputsOfDependentTask` as an input query.

```
open padv.builtin.task.MergeTestResults

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

When you run the **Merge Test Results** task, the build system runs this input query, which passes the current model and the outputs of the **Run Tests** task as inputs to the **Merge Test Results** task.

For each input, the build system checks for other artifacts that the inputs depend on by running the `InputDependencyQuery`. For the **Merge Test Results** task, the input dependency query allows the build system to find related artifacts, such as data dictionaries, that the model uses. Those dependencies can affect whether the task results are up-to-date or outdated.

Missing Dependencies

If the **I/O** column for a task is missing a dependency for your task, you can add that artifact by changing the `InputDependencyQuery` for the task. For example, in your process model:

```
addTask(pm,"MyCustomTask",...
    InputDependencyQuery = padv.builtin.query.FindArtifacts(...
    IncludePath="myDependency.m"));
```

In this example, a change to the file `myDependency.m` makes the results for the task `MyCustomTask` outdated.

Create Multiple Instances of Tasks

You can add multiple instances of a task to your process model to run different task configurations. For example, you can have one instance of the built-in task `padv.builtin.task.RunTestsPerModel` that runs normal mode tests and another instance that runs software-in-the-loop (SIL) tests.

When you create multiple instances of a task, there are a few considerations:

- Make sure that you assign each task instance a unique name, for example:

  ```
  milTask = pm.addTask(padv.builtin.task.RunTestsPerModel(...
          Name = "RunTestsNormalMode"));
  silTask = pm.addTask(padv.builtin.task.RunTestsPerModel(...
          Name = "RunTestsSILMode"));
  ```

  The build system uses the `Name` property as the unique identifier for the task.

- You can reconfigure the task instances to perform different functionalities. For example, starting in R2023a, you can run tests in different simulation modes without having to change the test definition:

```
milTask.SimulationMode = "Normal"; % supported starting in R2023a
silTask.SimulationMode = "Software-in-the-Loop"; % supported starting in R2023a
```

- You might need to reconfigure the task instances to avoid overwriting task outputs, for example:

```
% Specify normal mode outputs
milTask.OutputDirectory = defaultTestResultPath;
milTask.ReportName = '$ITERATIONARTIFACT$_Normal_Test';
milTask.ResultFileName = '$ITERATIONARTIFACT$_Normal_ResultFile';

% Specify SIL mode outputs
silTask.OutputDirectory = defaultTestResultPath;
silTask.ReportName = '$ITERATIONARTIFACT$_SIL_Test';
silTask.ResultFileName = '$ITERATIONARTIFACT$_SIL_ResultFile';
```

- You might need to reconfigure the input queries and iteration queries for tasks. For example, if you have multiple instances of `padv.builtin.task.RunTestsPerModel` and you want to merge the test results from both instances, you need to update the `InputQueries` for the task to get the outputs from both task instances:

```
%% Merge Test Results (Normal and SIL)
mergeTestTask = pm.addTask(padv.builtin.task.MergeTestResults(...
    InputQueries = [...
    padv.builtin.query.GetIterationArtifact,...
    padv.builtin.query.GetOutputsOfDependentTask(Task = "RunTestsNormalMode"),...
    padv.builtin.query.GetOutputsOfDependentTask(Task = "RunTestsSILMode")]));
```

For example code that shows how to create multiple instances of tasks inside the process model, see the following sections.

Run Tests in Normal and SIL Mode

Suppose that you want to have one instance of the built-in task `padv.builtin.task.RunTestsPerModel` that runs normal mode tests and another instance that runs software-in-the-loop (SIL) tests.

Starting in R2023a, in your process model, you can specify:

```
%% Run Tests in Normal Mode
    % Add task that runs tests in normal mode
    milTask = pm.addTask(padv.builtin.task.RunTestsPerModel(...
        Name = "RunTestsNormalMode",...
        Title = "Run Tests in Normal Mode"));
    milTask.SimulationMode = "Normal"; % supported starting in R2023a
    % Specify normal mode outputs
    milTask.OutputDirectory = defaultTestResultPath;
    milTask.ReportName = '$ITERATIONARTIFACT$_Normal_Test';
    milTask.ResultFileName = '$ITERATIONARTIFACT$_Normal_ResultFile';

    %% Run Tests in SIL Mode
    % Add task that runs tests in SIL mode
    silTask = pm.addTask(padv.builtin.task.RunTestsPerModel(...
        Name = "RunTestsSILMode",...
        Title = "Run Tests in SIL Mode"));
    silTask.SimulationMode = "Software-in-the-Loop"; % supported starting in R2023a
```

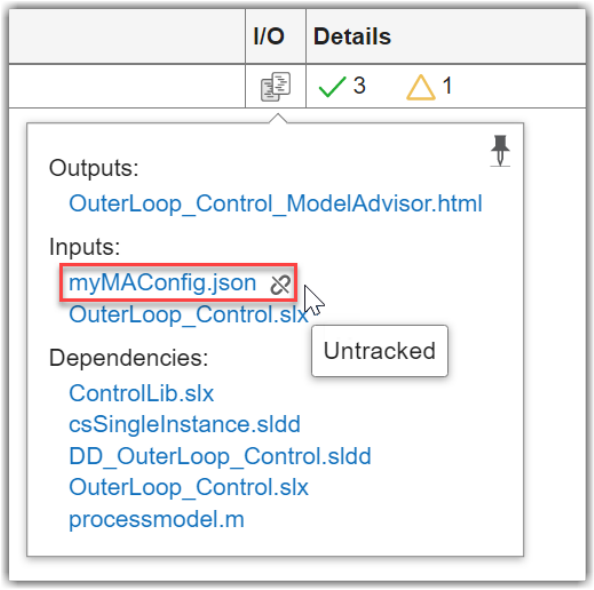
```
% Specify SIL mode outputs
silTask.OutputDirectory = defaultTestResultPath;
silTask.ReportName = '$ITERATIONARTIFACT$_SIL_Test';
silTask.ResultFileName = '$ITERATIONARTIFACT$_SIL_ResultFile';

%% Merge Test Results (Normal and SIL)
mergeTestTask = pm.addTask(padv.builtin.task.MergeTestResults(...
    InputQueries = [...
    padv.builtin.query.GetIterationArtifact,...
    padv.builtin.query.GetOutputsOfDependentTask(Task = "RunTestsNormalMode"),...
    padv.builtin.query.GetOutputsOfDependentTask(Task = "RunTestsSILMode")]));
% Specify Merged Test Outputs
% mergeTestTask.ReportPath = defaultTestResultPath;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Set Task relationships
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Set Task Dependencies
mergeTestTask.dependsOn(milTask,"WhenStatus",{'Pass','Fail'});
mergeTestTask.dependsOn(silTask,"WhenStatus",{'Pass','Fail'});
```

Turn Off Change Tracking for Input Artifacts

If you do not want the build system to mark a task as outdated when you make changes to a specific input artifact, you can turn off change tracking for that artifact by specifying the name-value argument `TrackArtifacts` as `false` when you use the built-in query `padv.builtin.query.FindFileWithAddress` to find the artifact. For example, the following process model code turns off change tracking for a Model Advisor configuration file, `myMAConfig.json`, used as an input for the **Check Modeling Standards** task:

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

When you run the task, the Process Advisor **I/O** column shows the outputs as **Untracked**. If you make a change to an untracked file, the build system does not mark the task as outdated.



By default, the build system marks any files outside the project as **Untracked** because you cannot track changes to files outside the project. The change tracking setting for the artifact is stored in the `Track` property in the `ArtifactAddress` property for the `padv.Artifact` object.

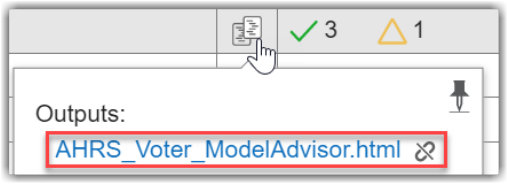
Note If you specify `TrackArtifacts=false`, you can no longer use that query object as an iteration query. The build system needs to track changes iteration artifacts to identify the iterations for the task.

Turn Off Change Tracking for Task Outputs

If you do not want the build system 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`. For example:

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

When you run the task, the Process Advisor **I/O** column shows the outputs as **Untracked**. If you make a change to an untracked file, the build system does not mark the task as outdated.



Define Task Relationships

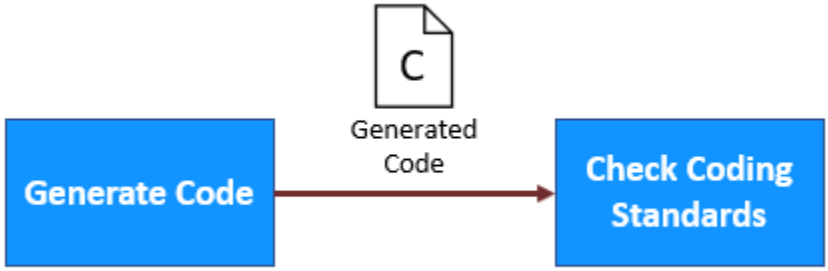
Task Relationships

When you author your process model, you might want to specify dependencies between tasks or specify a preferred task execution order. You can do this by adding a relationship between the tasks.

You can specify the relationship between two tasks as either a:

- `dependsOn` relationship — If a task should not run without another task running first, the task depends on the other task.

  For example, the **Check Coding Standards** task depends on the **Generate Code** task. Without the generated code, the **Check Coding Standards** task cannot run successfully.



  To specify that the generated code output by **Generate Code** task is the input for the **Check Coding Standards** task, the **Check Coding Standards** task uses the built-in query `padv.builtin.query.GetOutputsOfDependentTask` to find the outputs from the **Generate Code** task:

```
options.InputQueries = padv.builtin.query.GetOutputsOfDependentTask(...
    Task="padv.builtin.task.GenerateCode");
```

- `runsAfter` relationship — If a task does not depend on another task, but you want the task to run after that other task, the task should run after the other task.

  For example, the default process model specifies that the **Check Modeling Standards** task should run after the **Generate Simulink Web View** task. The **Check Modeling Standards** task can run successfully without the **Generate Simulink Web View** task. But the default process model specifies that, if possible, the build system should generate the web view before checking modeling standards.

For information on the `dependsOn` relationship, see "Specify Dependencies Between Tasks". For information on the `runsAfter` relationship, see "Specify Preferred Task Order".

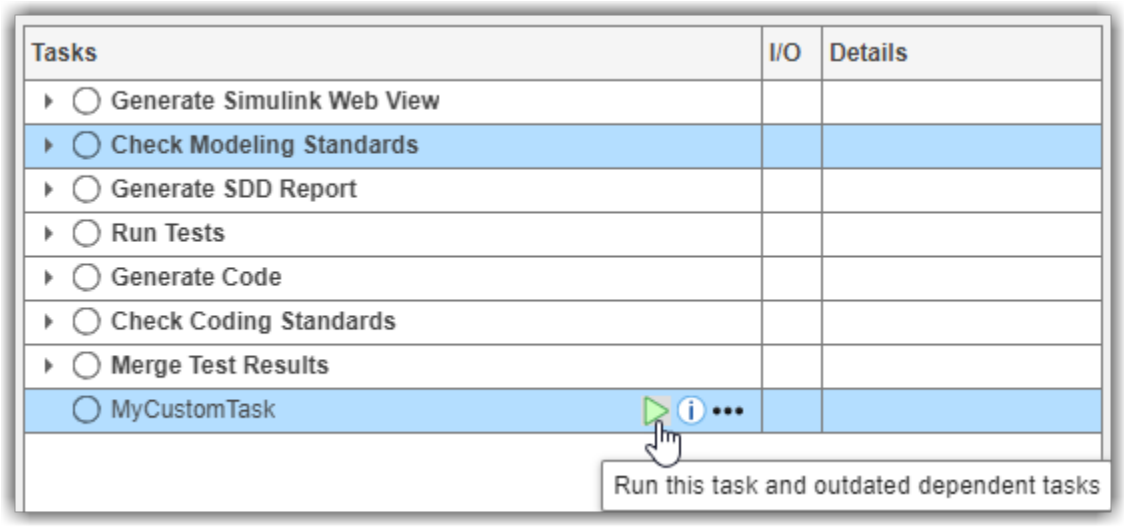
Specify Dependencies Between Tasks

You can use the `dependsOn` function in your process model to specify that a task depends on another task running first.

For example, to specify that a custom task, `MyCustomTask`, depends on the task **Check Modeling Standards**, use the `dependsOn` function on the task objects in your `processmodel.m` file:

```
% dependsOn(task,dependency)
dependsOn(taskObject,maTask);
```

If you open Process Advisor and point to a task that depends on another task, Process Advisor highlights the dependency.



If you try to run `MyCustomTask`, the build system will automatically run **Check Modeling Standards** first. By default, `MyCustomTask` will not run until **Check Modeling Standards** runs completely and returns a task status.

Note If you want to force a task to run independently, without dependent tasks running first, you can use the `Isolation` argument of `runprocess`:

```
runprocess(Tasks = taskName, Isolation = true)
```

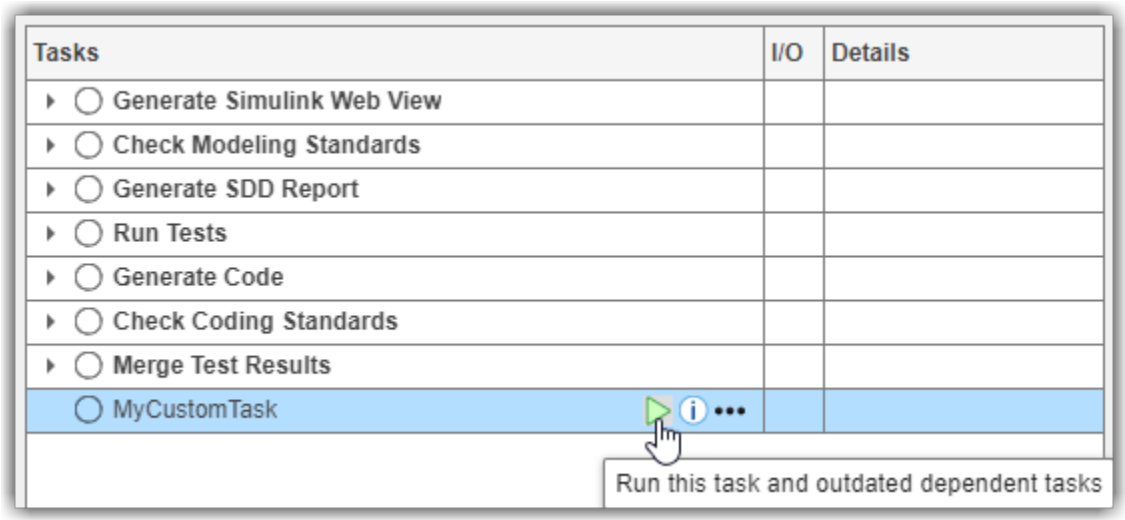
Specify Preferred Task Order

If a task does not depend on another task, but should run after that task, you can use the `runsAfter` function in your process model to specify your preferred task execution order. The build system will try to run the tasks in the order that you specify.

For example, to specify that a custom task, `MyCustomTask` (`taskObject`), should run after the **Generate Simulink Web view** task (`slwebTask`), you would add this code to the `processmodel.m` file:

```
% runsAfter(task,predecessors)
runsAfter(taskObject,slwebTask);
```

In Process Advisor, the tasks appears in the order that the build system will run them.



If a task **must always run** before another task, use `dependsOn` instead to make sure that both tasks always run together in sequence.

Note If you define multiple relationships between the same tasks, the build system only uses the most recent relationship and ignores previous relationships. For example, suppose you have a process model that contains:

```
runsAfter(taskA, taskB)
runsAfter(taskB, taskA) % build system only uses this relationship
```

This code defines a circular relationship between `taskA` and `taskB` because the code specifies both that `taskA` should run after `taskB` and that `taskB` should run after `taskA`.

By default, the build system ignores the first `runsAfter` command and only uses the second `runAfter` command.

If you want circular relationships to generate an error, specify the name-value argument `StrictOrdering` as `true`.

For example:

```
runsAfter(taskObject,slwebTask,...
    StrictOrdering = true); % error if this creates a circular relationship
```

Note By default, the build system only runs the predecessor tasks on artifacts that the task and the predecessor tasks have in common. If you need all task iterations of the predecessor tasks to run, specify `IterationArtifactMatching` as `false`.

For example:

```
runsAfter(taskObject,slwebTask,...
    IterationArtifactMatching = false); % run predecessor task on all its artifacts
```

Create Custom Task

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 steps that are not already covered by built-in tasks, you can 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".
- For more complex tasks — Create a MATLAB class that inherits from either a built-in task or the superclass `padv.Task` and then override class properties and methods to fit your needs. See "Custom Task for Specialized Functionality".

Custom Task that Runs Existing Script

If your custom task only needs to run an existing MATLAB script, you can 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. For example, inside 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");
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.

To define more complex custom tasks, use a MATLAB class instead. See "Create Task for Specialized Functionality".

Custom Task for Specialized Functionality

If your custom task only needs to run an existing MATLAB script, you can use the information in "Custom Task that Runs Existing Script". Otherwise, you should create and use a MATLAB class to define your custom task.

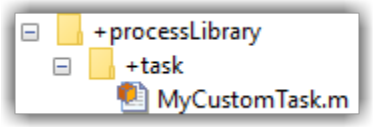
To create a task that performs a custom functionality:

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.
5. Add the task to your process model.

Create New MATLAB Class

Create a new MATLAB class in your project.

Tip Namespaces can help you organize the class definition files for your custom tasks. In the root of your project, 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:

- A built-in task — Use this approach if 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.
- 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.

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

```
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
```

This code uses the built-in task `padv.builtin.task.RunModelStandards`, but you can change those lines of code to use any built-in task.

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

```
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
```

This example 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 https://www.mathworks.com/help/matlab/matlab_oop/subclass-constructors.html.

Specify Task Properties

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

When you inherit from `padv.Task`, you must specify a `Name` (unique task identifier). Other class arguments are optional, but can help define the inputs and other properties of the task. Common class arguments that you might want to specify include:

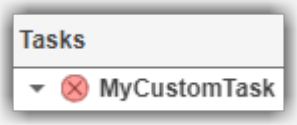
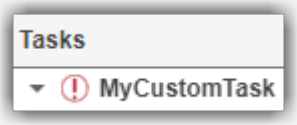
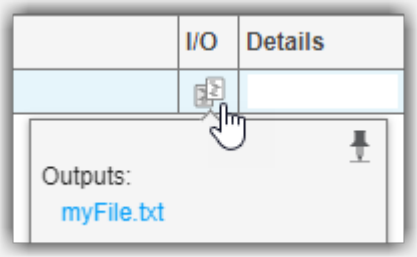
Argument	Description
`Name`	Unique identifier for task
`IterationQuery` (optional)	Which artifacts the task iterates over. For example, to have the task run one time for each model in the project, specify `IterationQuery` as the built-in query `"padv.builtin.query.FindModels"`. By default, custom tasks run once on the project. If you only want the task to run once for your project, do not specify an `IterationQuery`.
`InputQueries` (optional)	Inputs to the task. For example, to have the task run on each artifact that the task iterates over, specify the built-in query `"padv.builtin.query.GetIterationArtifact"`. The query `padv.builtin.query.GetIterationArtifact` returns the current artifact that the task is iterating over. The task results in the **I/O** column show the task inputs under **Inputs**.
`InputDependencyQuery` (optional)	Artifacts that inputs to the task depend on. The built-in tasks specify `InputDependencyQuery` as `padv.builtin.query.GetDependentArtifacts` to get the dependent artifacts for each task input. For example, if you specify a model as an input to a task and you specify `InputDependencyQuery` as `padv.builtin.query.GetDependentArtifacts`, the build system can find artifacts, such as data dictionaries, that the model uses. The task results in the **I/O** column show the task inputs under **Dependencies**.
`OutputDirectory` (optional)	Directory where the task outputs artifacts. If you do not specify `OutputDirectory` for a custom task, the build system stores task outputs in the `DefaultOutputDirectory` specified by `padv.ProcessModel`.

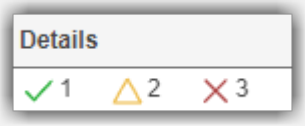
Keep or Override run Method

The `run` method defines the action that your custom task performs. 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 (with the iteration query `padv.builtin.query.FindModels`), 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)
    % "input" is a cell array of input artifacts
    % length(input) = number of input queries

    % identify model name
    model = input{1}; % get padv.Artifact from first input query
    [~,modelName,~] = fileparts(model.Address);
    % Load model
    obj.load_model(modelName);
    % <Perform task action using model>
    % Close model
    obj.close_model(modelName);

    % 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
```

The `run` method must return a `padv.TaskResult` object. Process Advisor and the build system use 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
`taskResult.Status = padv.TaskStatus.Pass`	
`taskResult.Status = padv.TaskStatus.Fail`	
`taskResult.Status = padv.TaskStatus.Error`	
`taskResult.OutputPaths=string(...` `fullfile("PA_Results","myFile.txt"));`	

`taskResult.ResultValues.Pass = 1;`
`taskResult.ResultValues.Warn = 2;`
`taskResult.ResultValues.Fail = 3;`
	

Use Custom Task in Process

Add your custom task to your process model by using the `addTask` function. For example:

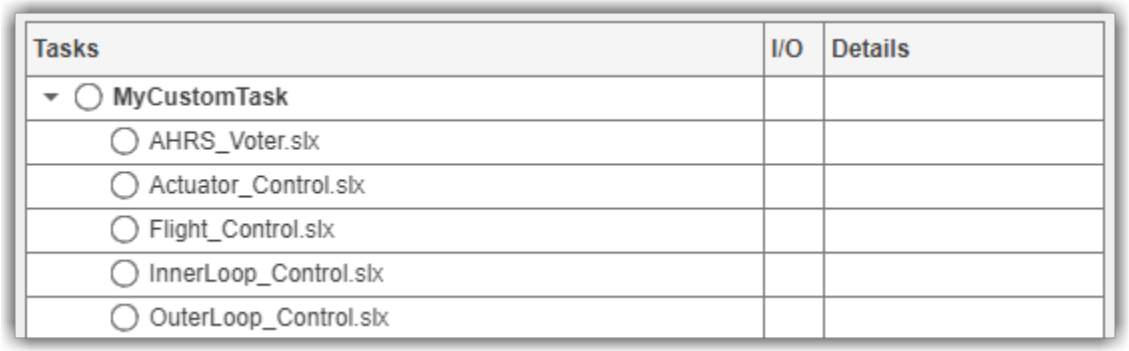
```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    addTask(pm,processLibrary.task.MyCustomTask);

end
```

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

You can confirm that your custom task is in the process by opening Process Advisor. In the MATLAB Command Window, enter:

```
processAdvisorWindow
```

The custom task, `MyCustomTask`, is in the **Tasks** column.



Run the task to confirm that the custom task runs and returns the expected status and results.

Note Certain MATLAB code requires a display to run successfully. 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 using your custom task.

For information, see "Set Up Virtual Display for No-Display Machine".

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 any 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 any of the checks generate 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 any 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 any warnings.

```
classdef MyRunModelStandards < padv.builtin.task.RunModelStandards
    % RunModelStandards, but use my Model Advisor configuration file
    % and fail the task if there are any 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 any 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 any 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:

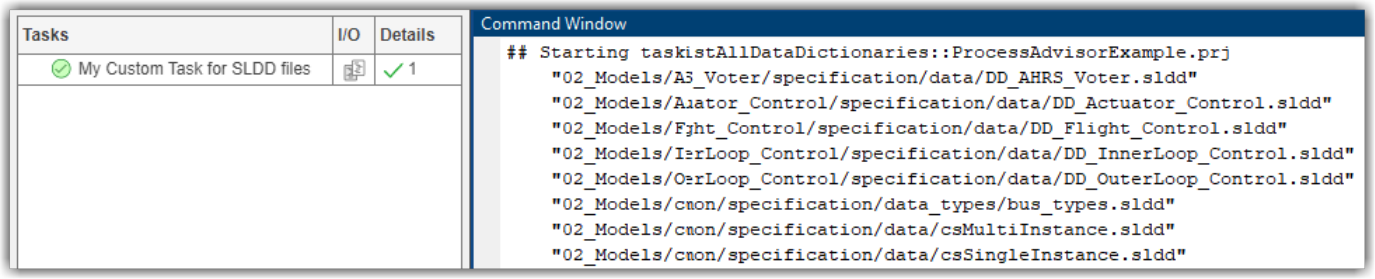
https://www.mathworks.com/help/matlab/matlab_oop/modifying-superclass-methods-and-properties.html

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}.Address]')
            taskResult = padv.TaskResult;
            taskResult.Status = padv.TaskStatus.Pass;
            taskResult.ResultValues.Pass = 1;
        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 addresses 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
Command Window
Starting taskistAllDataDictionaries::ProcessAdvisorExample.prj
"02_Models/A3_Voter/specification/data/DD_AHRS_Voter.sldd"
"02_Models/Aıator_Control/specification/data/DD_Actuator_Control.sldd"
"02_Models/Fght_Control/specification/data/DD_Flight_Control.sldd"
"02_Models/IrLoop_Control/specification/data/DD_InnerLoop_Control.sldd"
"02_Models/OrLoop_Control/specification/data/DD_OuterLoop_Control.sldd"
"02_Models/cmon/specification/data_types/bus_types.sldd"
"02_Models/cmon/specification/data/csMultiInstance.sldd"
"02_Models/cmon/specification/data/csSingleInstance.sldd"

Create Custom Query

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 whenever 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 if 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 any built-in query. Use the properties of

the query to specify which sets of artifacts you want the query to return. For examples, see the next section, "Example Custom Queries".

Note 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. For information, see the reference page for that built-in query in the "Built-In Query Library" in the Reference Book PDF.

- 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 array of `padv.Artifact` objects. For more information, see "padv.Artifact" in the Reference Book PDF.

3 You can test your custom query in the MATLAB Command Window executing the `run` function. For example, for a query `MyCustomQuery` saved in the namespace `processLibrary.query`:

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

Note that your project needs to be open for the query to find artifacts.

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:

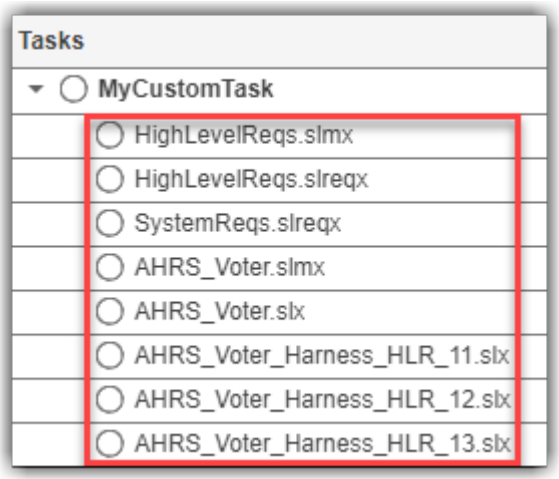
```
function processmodel(pm)
    % Defines the project's processmodel

    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

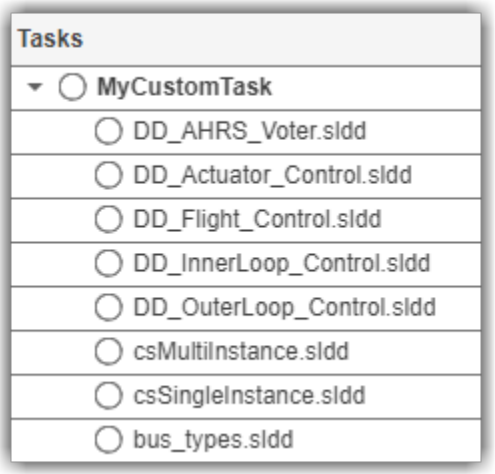
Suppose you want to find each of the data dictionaries in your project. There are no built-in queries that perform this functionality by default, but there is a built-in query `padv.builtin.query.FindArtifacts` that can find artifacts that meet certain search criteria. Effectively you can create your 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 the valid artifact types, see the "Artifact Types" chapter in the Reference Book PDF.

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
```



Hide File Extension in Process Advisor

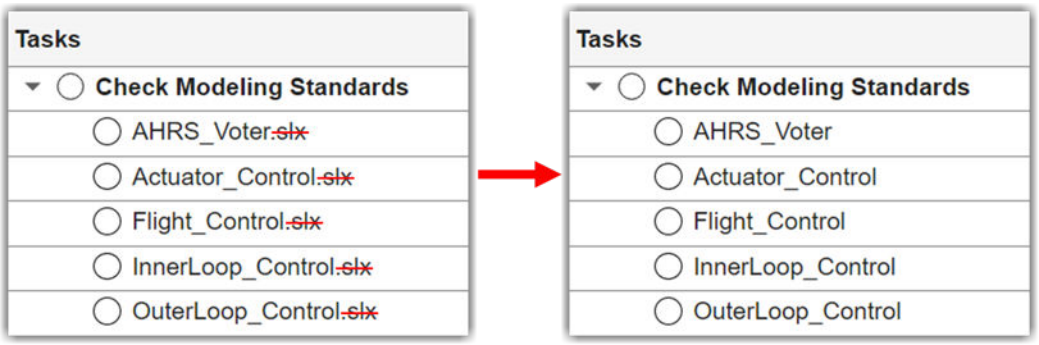
By default, the built-in query `padv.builtin.query.FindModels` returns the model file name and SLX extension. Suppose that you do not want to see the SLX extension in the Process Advisor user interface. You can create a custom query that removes the SLX extension from the `Alias` property:

```
classdef myCustomQuery < padv.builtin.query.FindModels
    methods
        function obj = myCustomQuery(NameValueArgs)
            arguments
                NameValueArgs.Name = "MyCustomQuery";
            end
        end
        function artifacts = run(obj,~)
            % call run method from FindModels
            artifacts = run@padv.builtin.query.FindModels(obj);
            % iterate over each Artifact and remove the file extension
            for i = 1:length(artifacts)
                if endsWith(artifacts(i).Alias,".slx")
                    artifacts(i).Alias = erase(artifacts(i).Alias,".slx");
                end
            end
        end
```

```
    end
end
```

If you use that custom query as the iteration query for a task, the task iterations no longer show the `.slx` file extension in the Process Advisor user interface.

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

    arguments
        pm padv.ProcessModel
    end

    %% Generate Simulink web view
    % Tools required: Simulink Report Generator
    slwebTask = pm.addTask(padv.builtin.task.GenerateSimulinkWebView());
    slwebTask.IterationQuery = processLibrary.query.myCustomQuery;

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
```

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.

Test Tasks and Queries

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` function 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.

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

```
processAdvisorExampleStart
```

2 Suppose that you want to test the query `padv.builtin.query.FindModels`. You can create an instance of this query. In the MATLAB Command Window, enter:

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

3 To see which artifacts the query returns, run the query.

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

```
artifacts =

  1×5 Artifact array with properties:

    Type
    Parent
    ArtifactAddress
```

In this example, the query returns the five models in the example project.

Tip If you open the `ArtifactAddress` property, you can see the names of each of the models returned by the `padv.builtin.query.FindModels` query.

```
artifacts.ArtifactAddress
```

---

4 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");
```

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

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

```
artifacts =

  Artifact with properties:

                Type: "sl_model_file"
              Parent: [0×0 padv.Artifact]
     ArtifactAddress: [1×1 padv.util.ArtifactAddress]
```

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

```
artifacts.ArtifactAddress

ans =

ArtifactAddress

          FileAddress: "02_Models/AHRS_Voter/specification/AHRS_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.

6 Suppose that you want to run the task `padv.builtin.task.GenerateSimulinkWebView` on the `AHRS_Voter` model returned in `artifacts`. You can run that 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))
```

Tip 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.

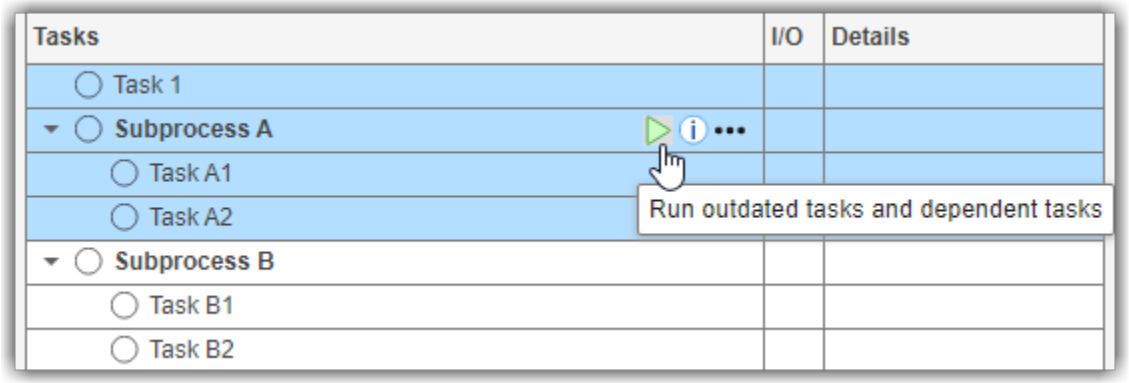
```
runprocess(...
Tasks = "padv.builtin.task.GenerateSimulinkWebView",...
FilterArtifact = artifacts(1),...
Force = true,...
Isolation = true)
```

For more information, see "runprocess" in the Reference Book PDF or, in the MATLAB Command Window, enter:

```
help runprocess
```

Group Tasks Using Subprocesses

You can use a subprocess to group related tasks, create a hierarchy of tasks, and share parts of a process. A *subprocess* is a self-contained sequence of tasks, inside a process or other subprocess, that can run standalone.



To group the tasks in your process model:

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

   ```
   spA = pm.addSubprocess("Subprocess A");
   ```

2. Add your tasks directly to the subprocess by using `addTask`.

   ```
   tA1 = spA.addTask("Task A1");
   tA2 = spA.addTask("Task A2");
   ```

   Note You do not need to add the task to both the subprocess and process model.

3. Specify the relationship between the tasks and subprocesses in your process.

   You can use the `dependsOn` and `runsAfter` functions to define the relationships.

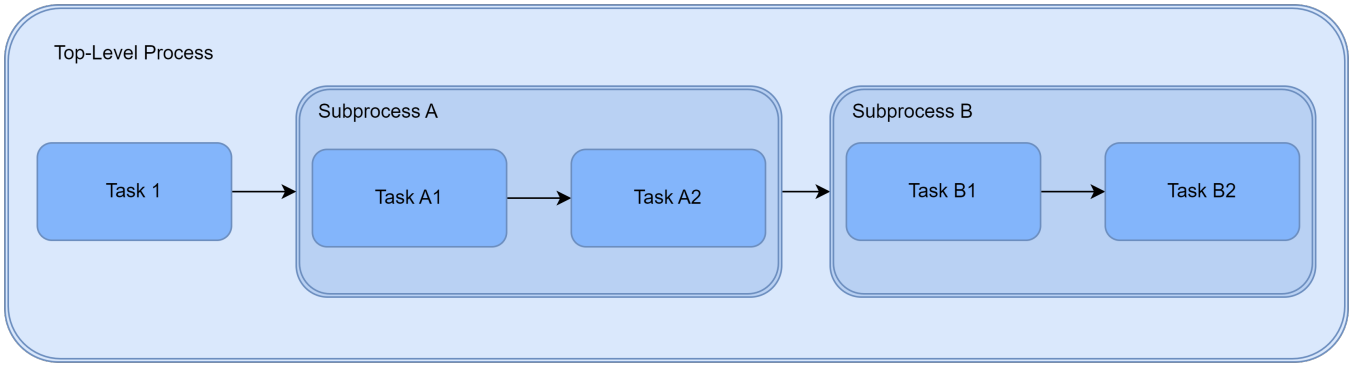
   For example, the following process model defines a process in which `Task 1` runs, then `Subprocess A`, and then `Subprocess B`.

   ```
   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
```

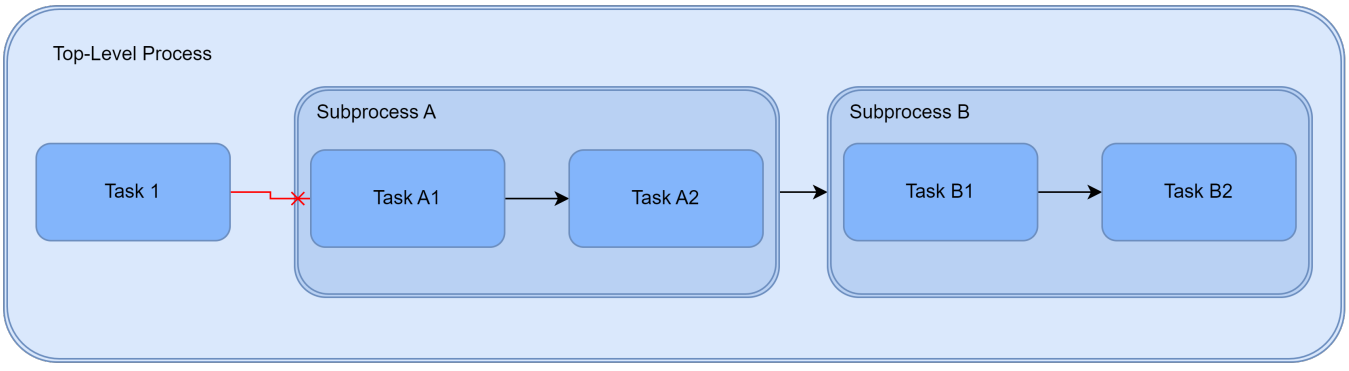
The build system executes each of the tasks inside a subprocess before existing the subprocess.

The following diagram shows a graphical representation of the relationships defined by that process model.



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.



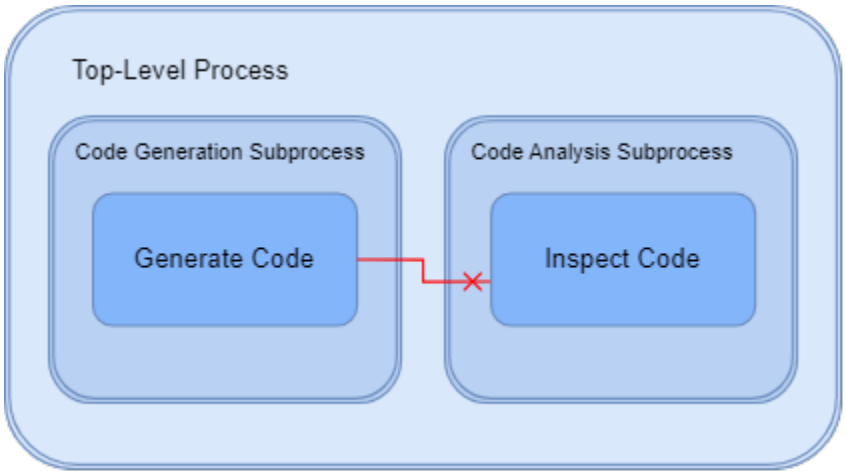
In this case, you need to create a relationship between the `Task 1` and `Subprocess A` instead.

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.

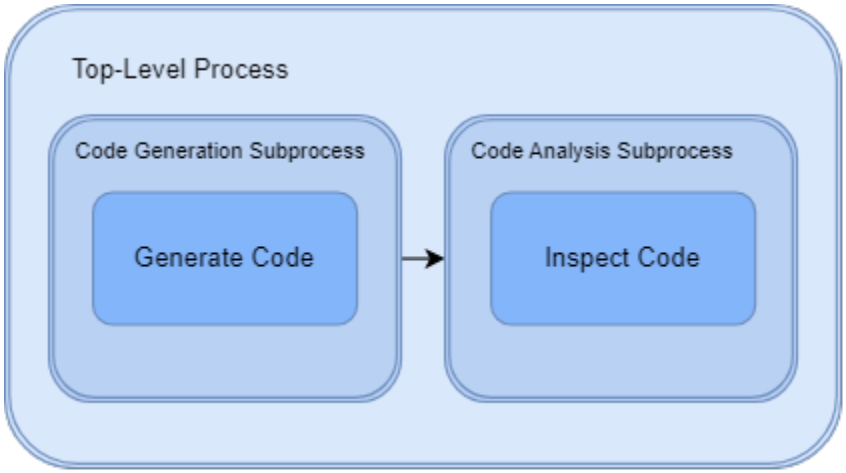


If you try to have a code analysis task in one subprocess depend on a code generation task in another subprocess, Process Advisor generates an error like: `Invalid dependency between Task 'padv.builtin.task.RunCodeInspection' and 'padv.builtin.task.GenerateCode'. Make sure 'padv.builtin.task.GenerateCode' 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 any code analysis tasks 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);
```


Top-Level Process
Code Generation Subprocess
Generate Code
Code Analysis Subprocess
Inspect Code

Example Process Models

Add One Built-In Task and One Custom Task

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

     % Adding a built-in task
     task1 = addTask(pm,padv.builtin.task.RunModelStandards);

     % Adding a custom task
     task2 = addTask(pm,"Custom Task",Action=@CustomAction);

     % Specify that the custom task should run after the built-in task
     runsAfter(task2,task1);

end

 function results = CustomAction(~)
     disp("Hello, world")
     results = padv.TaskResult;
 end
```

Specify a Task Execution Order

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

     %% ADD CUSTOM TASKS TO THE PROCESS MODEL
     task1 = addTask(pm,"Task 1");
     task2 = addTask(pm,"Task 2");
     task3 = addTask(pm,"Task 3");
     task4 = addTask(pm,"Task 4");
     task5 = addTask(pm,"Task 5");

     %% SPECIFY THE TASK EXECUTION ORDER
     % task2 must run after task1
         runsAfter(task2,task1,StrictOrdering=true);
     % task3 should run after task2
     % but task3 can run independently
         runsAfter(task3,task2);
     % task4 should run after task3
     % but task4 can run independently
         runsAfter(task4,task3);
     % task5 must run after task4
         runsAfter(task5,task4,StrictOrdering=true);

end
```

Include Multiple Instances of a Task

If you include duplicates of a task, the Process Advisor will return an error:`Invalid definition in 'processmodel.m' file. Unable to add task because a task named` *`taskName`* `already exists.`

To include multiple instances of the same type of task, you need to specify different values of `Name` for each of the tasks. For built-in tasks, you need to override the `Name` when you create the task iteration.

For example, suppose you want to add two versions of the built-in task `padv.builtin.task.RunTestsPerTestCase`. When you create an instance of the task by using `padv.builtin.task.RunTestsPerTestCase`, you need to specify a different value for the `Name`.

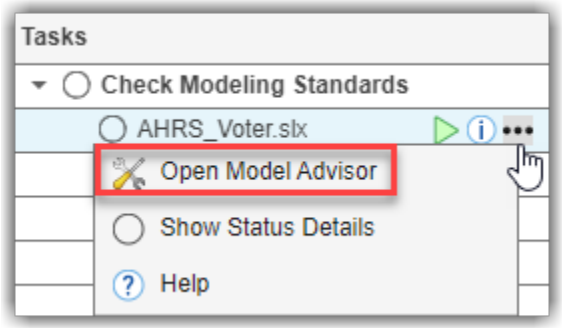
```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end
    taskA_v1 = addTask(pm,...
        padv.builtin.task.RunTestsPerTestCase(Name="Something else"),...
        Title="Task A - Version 1");
    taskA_v2 = addTask(pm, padv.builtin.task.RunTestsPerTestCase,...
        Title="Task A - Version 2");
end
```

You can then specify different values for the `IterationQuery` so that the tasks operate on different sets of artifacts.

For more information, see "Create Multiple Instances of Tasks" in this PDF.

Specify Which Tool to Launch for a Custom Task

When you point to a task in the Process Advisor app, you can click the ellipsis (**...**) to see more options. For built-in tasks, you have the option to launch a tool 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.



For custom tasks, you can specify the property `LaunchToolAction` to associate a tool with the options menu for the task.

For example, suppose you have a custom task that runs on each model in the project and you want the task to launch the Dependency Analyzer for the model. For `LaunchToolAction`, specify the handle to a function that launches the tool.

```
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=@myLaunchToolAction);

end

function result = myLaunchToolAction(obj, artifact)

     result = struct('ToolLaunched', false);

     % identify model name
     [~,modelName,~] = fileparts(artifact.Address);

     % open Dependency Analyzer for model
     depview(modelName)

     result.ToolLaunched = true;

end
```

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`.

Note Although you can launch other tools from the Process Advisor app, make sure you use the Process Advisor app or build system to run your tasks and to collect task results. The app and build system might not detect changes to settings, files, or task results from actions that you perform in other tools.

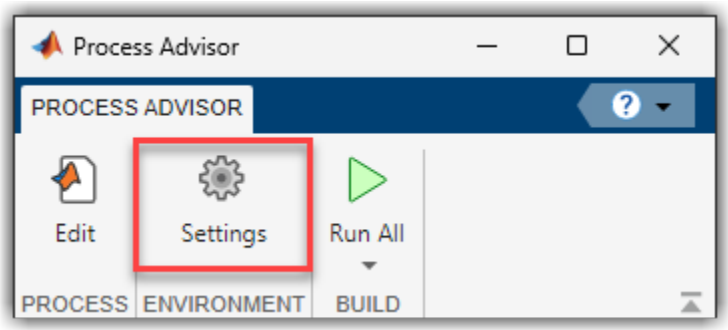
Control Builds

This chapter describes how to run builds and customize build execution:

Specify Settings for Builds

There are several settings that you can use to customize the behavior of the build system. These behaviors impact how the Process Advisor app and `runprocess` function run tasks. 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** — These settings are stored in the project and are shared with everyone using this project.
- **User Settings** — These settings only apply to the current user.

Project Settings

Setting	Usage
Incremental build	Select this setting to allow the build system to automatically detect changes and mark task results as outdated. **Default:** On
Enable model caching	Select this setting to allow the build system to cache models during builds. **Default:** Off
Suppress outputs to command window	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. **Default:** Off

Setting	Usage
Show file extensions	Select this setting to show file extensions for all task iteration artifacts in the **Tasks** column in Process Advisor. To keep file extensions in the results for a specific query, you can specify the query property `ShowFileExtension` as `true`. For information, see `padv.Query` in the Reference Book PDF. **Default:** Off

For more information about a specific setting, see `padv.ProjectSettings` in the Reference Book PDF.

User Settings

Setting	Usage
Detect duplicate outputs	Select this setting to allow the build system to generate an error message when multiple tasks attempt to write to the same output file. **Default:** On
Garbage collect task outputs	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. **Default:** On
Show detailed error messages	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. **Default:** Off
Add process model as dependency	Select this setting to add the process model file as a dependency. 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. If you do not want changes to the process model to make task statuses and task results outdated, clear this setting. **Default:** On

For more information about a specific setting, see `padv.UserSettings` in the Reference Book PDF.

Build System API Overview

Run Tasks in Pipeline

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

Run All Tasks

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

```
runprocess()
```

Run Specific Task

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 Tasks for Specific Artifact

To only run the tasks 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 "runprocess" in the Reference Book PDF.

View Available Tasks in Pipeline

- Use the `generateProcessTasks` function to return a list of the available tasks in the current process model.

  ```
  generateProcessTasks
  ```

- List a set of specific tasks by using the `FilterArtifact` argument. For example, you can specify the relative path to a model and list the associated tasks.

  ```
  % 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

For example, if you run the tasks in the default MBD pipeline, the report provides an overview of the:

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

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`.

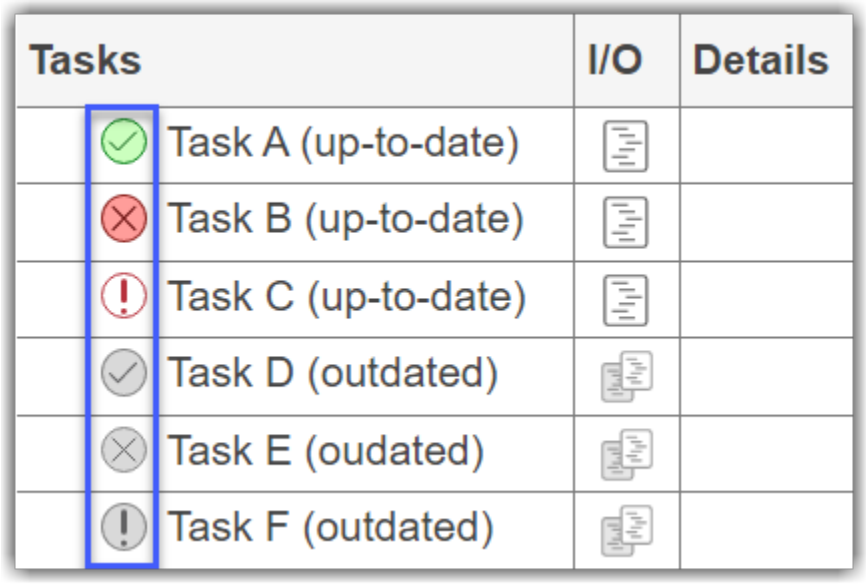
```
htmlReport=padv.ProcessAdvisorReportGenerator(Format="html-file");
generateReport(htmlReport);
```

Note Alternatively, you can specify `GenerateReport` as `true` when you use `runprocess`: `runprocess(GenerateReport = true)`.

Incremental Builds

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.

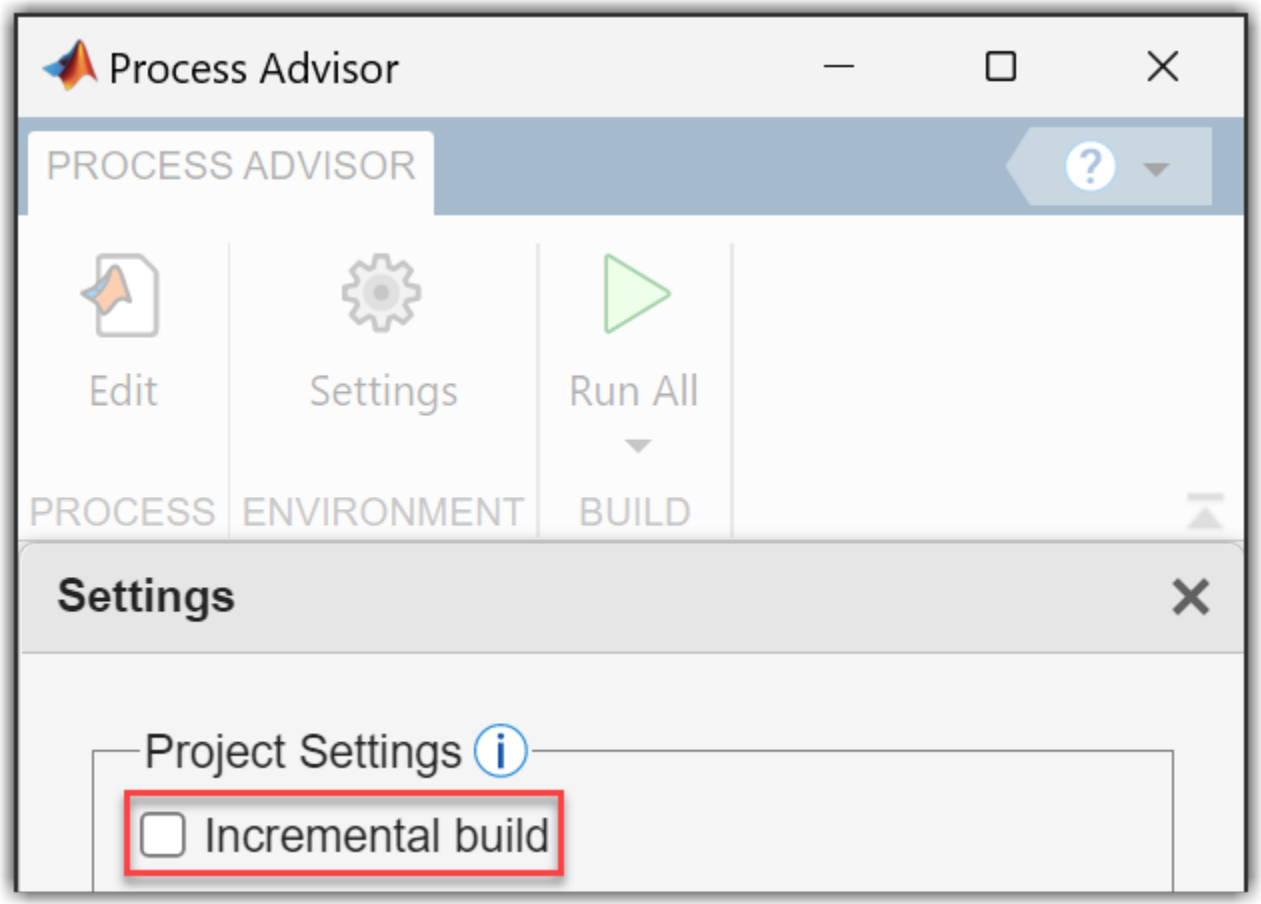
When incremental builds are enabled, the task status icons in the **Tasks** column indicate whether the task results are up-to-date or outdated. Up-to-date task results have task status icons that are green for tasks that pass and red for tasks that fail or generate errors. Outdated task results have task status icons that are gray.



How to Disable Incremental Builds

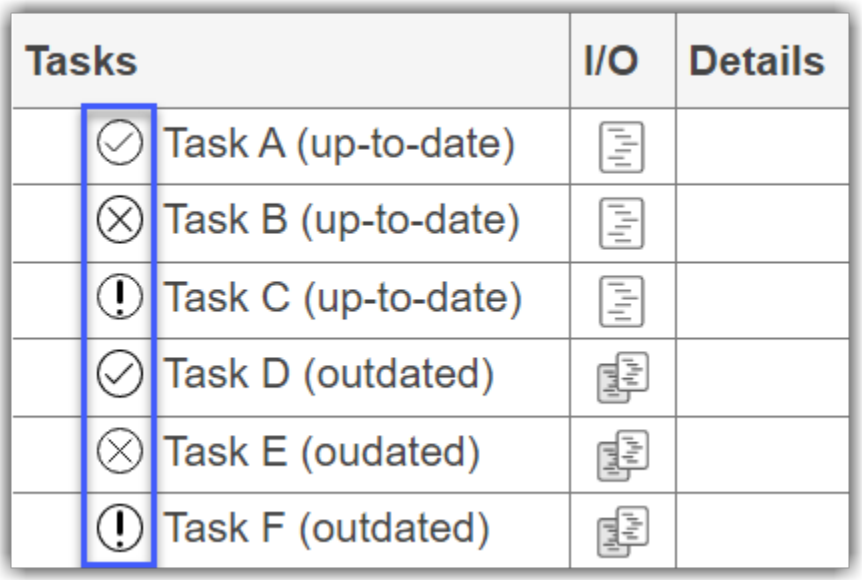
If you want to force the build system and the Process Advisor app to re-run task iterations, you can disable incremental builds for the project. When you disable incremental builds, the build system and the Process Advisor app do not identify any results as up-to-date or outdated, and effectively force run task iterations in the project.

To disable incremental builds, open the Process Advisor app and, in the toolstrip, click **Settings** and clear **Incremental build**.



The incremental build setting is stored in the project and is shared with everyone using the project.

When incremental builds are disabled, the task status icons in the **Tasks** column appear in black and white because the app is no longer identifying up-to-date or outdated results. These statuses only indicate whether the task passed, failed, generated an error, or did not run.


Tasks
I/O
Details
Task A (up-to-date)
Task B (up-to-date)
Task C (up-to-date)
Task D (outdated)
Task E (oudated)
Task F (outdated)

Best Practices for Effective Builds

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 "Incremental Builds" section in this PDF and the documentation for the `runprocess` function in the Reference Book PDF.

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` in the Reference Book PDF.

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
```

Integrate into CI

This chapter describes how to integrate MathWorks tools into a CI system using the support package CI/CD Automation for Simulink Check:

Prerequisites

Before integrating with a CI system:

1. Check that the CI system can run MATLAB. For information on the supported platforms, see https://www.mathworks.com/help/matlab/matlab_prog/continuous-integration-with-matlab-on-ci-platforms.html.

   Note **License Considerations for CI:** If you plan to perform CI on many hosts or on 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).

2. Install the support package CI/CD Automation for Simulink Check on the MATLAB instance or instances that run in your CI system. For information on how to use the support package with Docker, see "Create Docker Container for Support Package".
3. Certain MATLAB code, including some built-in tasks, requires a display to run successfully. 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:

   - **Generate SDD Report**
   - **Generate Simulink Web View**
   - **Generate Model Comparison**

   For information, see "Set Up Virtual Display for No-Display Machine".

For related information on how CI/CD can apply to model-based design, see https://www.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models.html.

Approaches to Pipeline Configuration

A *pipeline* is a collection of automated procedures and tools that execute in a specific order to enable a streamlined software delivery process. CI systems allow you to define and configure a pipeline by using a pipeline configuration file.

When integrating MBD projects into CI, there are three main approaches to creating and maintaining 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 no longer 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 no longer have to manually write or generate pipeline configuration files, but setting up a branching workflow can be complex.

The following table lists which approaches are supported on each CI system.

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

The next sections provide information on how to integrate into specific CI platforms using a recommended approach.

Integrate into GitHub

With the pipeline generator, you can generate a pipeline configuration file that you can use to define a GitHub Actions workflow.

Tip To see an example project that uses an example pipeline configuration file, open the GitHub example project. In the MATLAB Command Window, enter:

```
processAdvisorGitHubExampleStart
```

1. In MATLAB, configure your project to use local Git™ source control. Open your project and, on the **Project** tab, click **Use Source Control**. In the Source control Information dialog box, click **Add Project to Source Control**. In the Add to Source Control dialog box, in the **Source control tool** list, select `Git` and then click **Convert**.

2. In GitHub, create a private GitHub repository. For information, see the GitHub documentation: https://docs.github.com/en/get-started/quickstart/create-a-repo

3. Create a self-hosted runner. For information, see the GitHub documentation: https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/adding-self-hosted-runners

4. In MATLAB, on the **Project** tab, click **Remote** and specify the URL for the remote origin in GitHub where your repository is located. For more information, see https://www.mathworks.com/help/simulink/ug/add-a-project-to-source-control.html.

5. In your local MATLAB installation, configure the pipeline generation settings. In the Command Window, create a `padv.pipeline.GitHubOptions` object and specify the location of your MATLAB installation for your runner.

   The object stores the settings 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, to 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";
   ```

   Note If you use Git submodules to organize your projects and you want the generated pipeline configuration file to automatically checkout your Git submodules at the beginning of each stage of the pipeline, specify the property `CheckoutSubmodules` as either `"true"` or `"recursive"`. For more information, see `padv.pipeline.GitHubOptions` in the Reference Book PDF.

6. 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 `GeneratedYMLFileName` and `GeneratedPipelineDirectory` properties of the `padv.pipeline.GitHubOptions` object control the name and location of the generated pipeline configuration file.

The generated pipeline configuration file makes use of the following GitHub Actions:

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

To use the generated pipeline configuration file in your GitHub repository, you need to create a workflow. For general information on how to create a GitHub Actions workflow, see: https://docs.github.com/en/actions/quickstart#creating-your-first-workflow.

7 Open the generated pipeline configuration file and copy the file contents.

8 In GitHub, create a new directory, `.github/workflows`.

9 Inside `.github/workflows`, create a new YAML file, `github-actions-demo.yml`.

10 Paste the contents of `simulink_pipeline.yml` inside the `github-actions-demo.yml` file.

11 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.

How Automatic Pipeline Generation Works

A *pipeline* is a collection of automated procedures and tools that execute in a specific order to enable a streamlined software delivery process. CI systems allow you to define and configure a pipeline by using a pipeline file.

- In GitLab, you can configure your pipeline by using a `.yml` file that you store in your project. The `.yml` file can configure different parts of your CI/CD jobs including the stages of the job, the tag for your GitLab Runner, the script that the Runner executes, and artifacts you want to attach to a successful job. The support package contains an example pipeline configuration file that you can use in your project.
- In Jenkins, you can 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 contains an example pipeline configuration file, `Jenkins`, that you can use in your project.

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 automatically generate the updated pipeline configuration files for you. After you do the initial setup for the pipeline generator, you no longer 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 automatically generate any necessary pipeline configuration files for you.

You can automatically generate pipeline configuration files on these CI platforms:

- GitLab
- Jenkins

Initial Setup

The major steps to set up the pipeline generator are:

1. Connect your MATLAB project to either a GitLab or Jenkins project.
2. Add the example pipeline configuration file to your project.
3. Edit the example pipeline configuration file to specify any credentials or other information needed to run jobs in your CI system.
4. Optionally, you can edit the example pipeline configuration file to change how the pipeline generator creates and executes pipelines in CI.
5. Push the changes to your source control system. By default, GitLab projects use `.gitlab-ci.yml` as the pipeline configuration file and Jenkins projects use `Jenkinsfile` as the pipeline configuration file.

For instructions, see either:

- "Integrate into GitLab"
- "Integrate into Jenkins"

Automatically Generated Pipelines

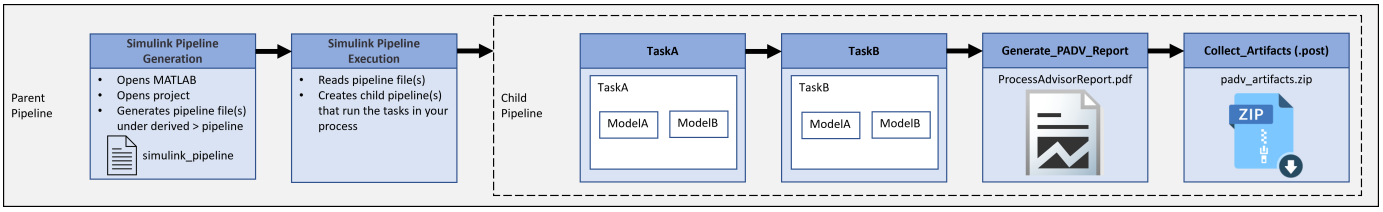
After you perform the initial setup and trigger your pipeline, the pipeline generator generates a parent pipeline and a child pipeline.

The parent pipeline contains two stages:

- **Simulink Pipeline Generation** — This stage analyzes your project and process model to automatically generate the necessary pipeline configuration files to run your process in CI. The main, generated pipeline configuration file is called `simulink_pipeline.yml` in GitLab or `simulink_pipeline` in Jenkins. If you want to view any of 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`.



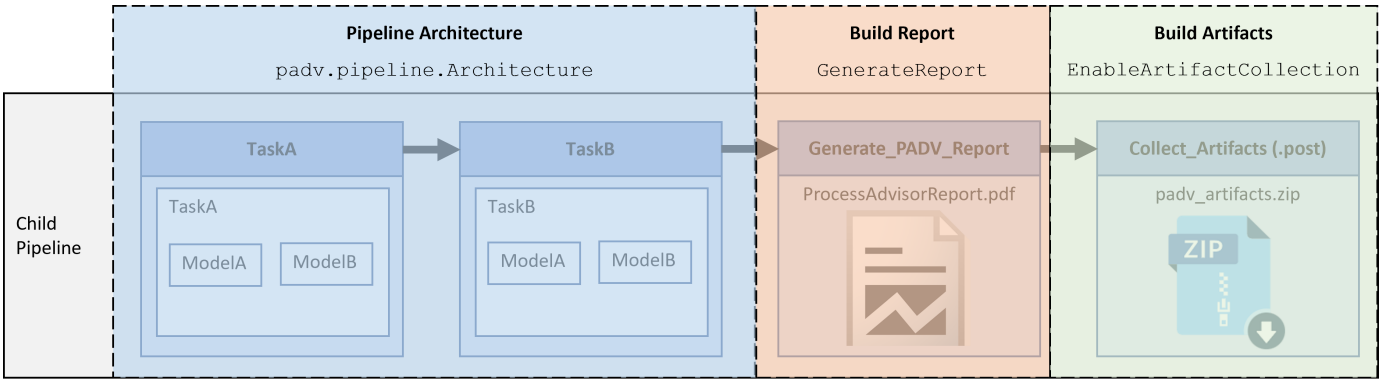
Optional Pipeline Customization

You can run the pipeline generator using the default settings 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. For GitLab, the CI options object is `padv.pipeline.GitLabOptions`. For Jenkins, the CI options object is `padv.pipeline.JenkinsOptions`.

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".

For more information, see `padv.pipeline.GitLabOptions` or `padv.pipeline.JenkinsOptions` in the Reference Book PDF.

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".

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")
```

For more information, "Integrate into GitLab" or "Integrate into Jenkins".

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 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;
  ```

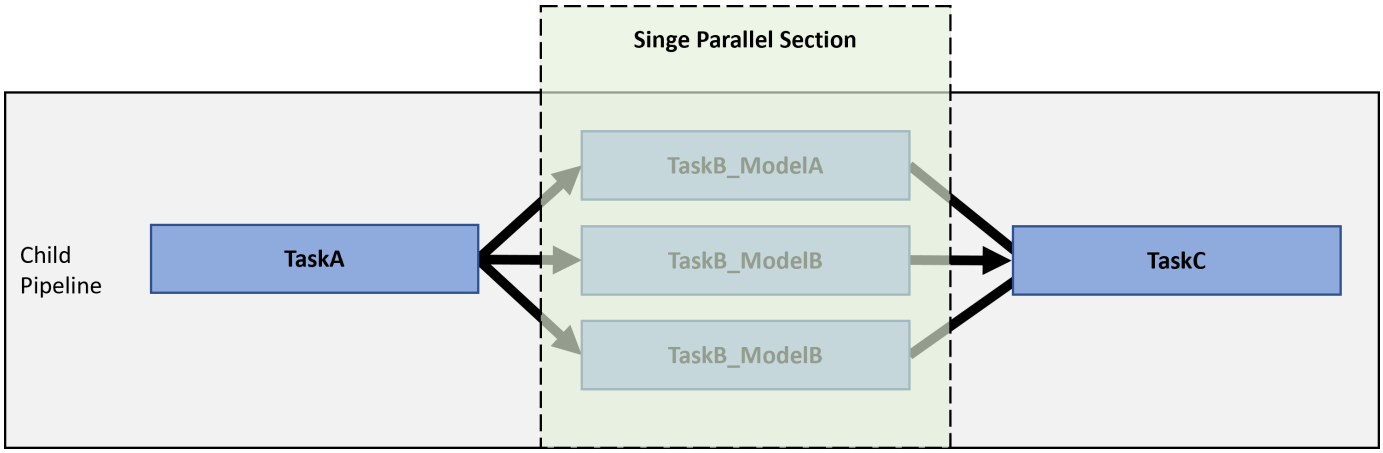
- Split any code analysis tasks into two tasks. One task for reference models and one task for top models. The task for top models must iterate over the project file. The built-in code analysis tasks, like `padv.builtin.task.RunCodeInspection`, are able to unpack the code generation target from the external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

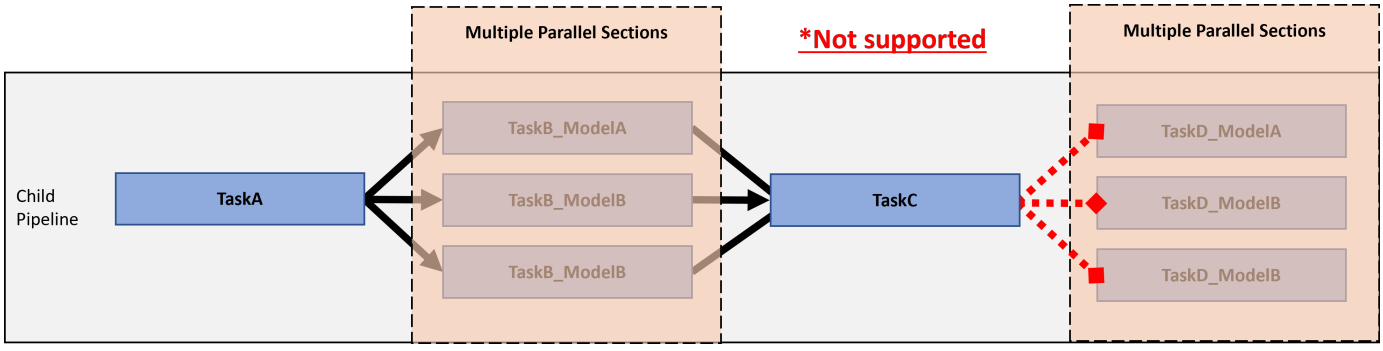
```
        '$INPUTARTIFACT$');
slciTopTask.OutputDirectory = string(fullfile('$DEFAULTOUTPUTDIR$','code_inspection'))
```

- 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.

Integrate into GitLab

A *pipeline* is a collection of automated procedures and tools that execute in a specific order to enable a streamlined software delivery process. CI systems allow you to define and configure a pipeline by using a pipeline file. In GitLab, you can configure your pipeline by using a `.yml` file that you store in your project. The `.yml` file can configure different parts of your CI/CD jobs including the stages of the job, the tag for your GitLab Runner, the script that the Runner executes, and artifacts you want to attach to a successful job.

The support package CI/CD Automation for Simulink Check comes with an example pipeline configuration file that you can add to your project to automatically run pipelines in GitLab. The example file uses the pipeline generator to generate and execute pipelines for you so that you do not need to manually update any pipeline files when you change the tasks and artifacts in your project.

Tip To see an example project that uses the example pipeline configuration file, open the GitLab example project. In the MATLAB Command Window, enter:

```
processAdvisorGitLabExampleStart
```

Integrate Using Default Options

1. In MATLAB, configure your project to use local Git source control. In MATLAB, on the **Project** tab, click **Use Source Control**. In the Source control Information dialog box, click **Add Project to Source Control**. In the Add to Source Control dialog box, in the **Source control tool** list, select `Git` and then click **Convert**.

2. In GitLab, set up a remote GitLab repository by creating a new blank project. For information, see the GitLab documentation: https://docs.gitlab.com/ee/

3. Install, register, and start a GitLab Runner. For information, see the GitLab documentation: https://docs.gitlab.com/runner/install/index.html

4. In MATLAB, on the **Project** tab, click **Remote** and specify the URL for the remote origin in GitLab where your repository is hosted. For more information, see https://www.mathworks.com/help/simulink/ug/add-a-project-to-source-control.html.

5. Change your current folder to your project root and copy the example pipeline configuration file into your project.

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

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

   Note The example pipeline configuration file is generic and can work with any project.

6. Open and inspect the example pipeline configuration file, `.gitlab-ci.yml`, in your project.

   The file `.gitlab-ci.yml` defines a parent pipeline. The parent pipeline uses the pipeline generator, `padv.pipeline.generatePipeline`, to automatically generate and execute a child

pipeline for your project. The options for the child pipeline are specified by the object `padv.pipeline.GitLabOptions`. For more information about parent-child pipelines, see https://docs.gitlab.com/ee/ci/pipelines/downstream_pipelines.html.

7 In your `.gitlab-ci.yml` file, replace `padv_demo_ci` with the CI/CD tag associated with your GitLab Runner.

For example, if your Runner is associated with the tag `high_memory`, change the `tags` field to:

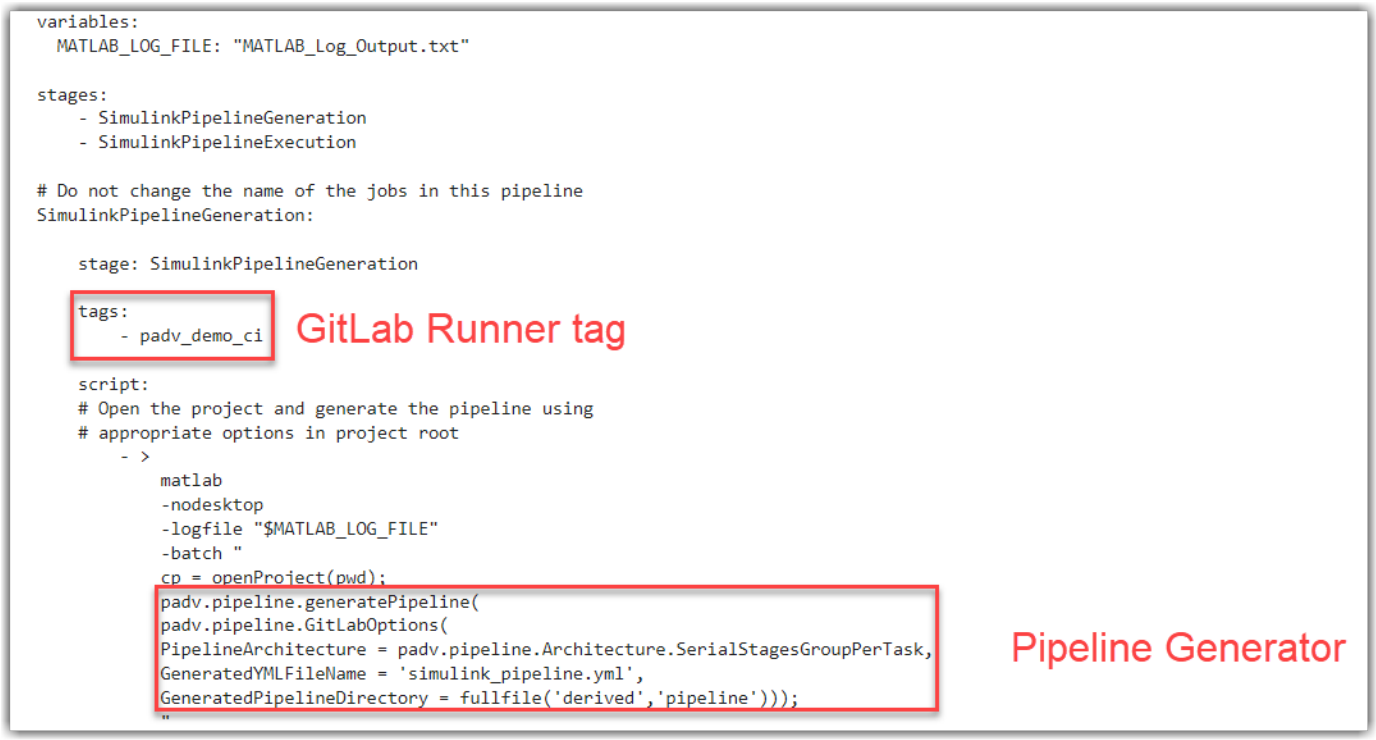
```
tags:
    - high_memory
```

8 Modify the object `padv.pipeline.GitLabOptions` to specify the CI/CD tag associated with your GitLab Runner. `.gitlab-ci.yml` passes the tag to the child pipeline.

For example, if your Runner is associated with the tag `high_memory`, you would specify:

```
padv.pipeline.generatePipeline(
padv.pipeline.GitLabOptions(
Tags='high_memory',
PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask,
GeneratedYMLFileName = 'simulink_pipeline.yml',
GeneratedPipelineDirectory = fullfile('derived','pipeline')));
```

Now your `.gitlab-ci.yml` file will have your GitLab Runner tag specified in the `tags` field and in your `padv.pipeline.GitLabOptions` in the call to the pipeline generator function `padv.pipeline.generatePipeline`.



9 Add your `.gitlab-ci.yml` file to your project.

10 Push the changes to your GitLab repository.

By default, a GitLab project automatically considers any file named `.gitlab-ci.yml` as the CI/CD configuration file for the repository. Your GitLab Runner can now automatically generate and execute a custom pipeline for your project each time that you submit changes.

Note You do not need to update the `.gitlab-ci.yml` file if you make changes to your projects or process model. The pipeline generator generates the child pipeline 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.

If you use Git submodules to organize your project, note that by default, the example pipeline configuration file recursively fetches the Git submodules. If you do not want this behavior, change the `GIT_SUBMODULE_STRATEGY` in the YAML file to `none`:

```
GIT_SUBMODULE_STRATEGY: none
```

In GitLab, your pipeline will contain two upstream jobs:

- **SimulinkPipelineGeneration** — Generates a child pipeline file.
- **SimulinkPipelineExecution** — Executes the child pipeline file. By default, the child pipeline contains these downstream jobs:
  - One job for each task defined in the process model file
  - One job, `Generate_PADV_Report`, that generates a Process Advisor build report
  - One job, `Collect_Artifacts`, that collects build artifacts

The pipeline generator automatically generates JUnit-style XML reports for each task. When you open the **SimulinkPipelineExecution** job in GitLab, the **Tests** tab shows a summary of the task results. For information on how JUnit information appear in GitLab, see the GitLab documentation: https://docs.gitlab.com/ee/ci/testing/unit_test_reports.html#view-unit-test-reports-on-gitlab. If you do not want to generate JUnit reports, specify the `GenerateJUnitForProcess` property in `padv.pipeline.GitLabOptions` as `false`.

If you want to change how the downstream jobs get organized and executed, you can modify the properties of the `padv.pipeline.GitLabOptions`. For example, you can modify the `PipelineArchitecture` property to change the number of stages and the grouping of tasks in each stage of the child pipeline. For more information, see "Customize Child Pipeline" or enter this code in the MATLAB Command Window:

```
help padv.pipeline.GitLabOptions
```

Customize Child Pipeline

You can use the properties of `padv.pipeline.GitLabOptions` to control which GitLab Runner tags to associate with the child pipeline, the number of stages and the grouping of tasks in the child pipeline (defined by the pipeline architecture), how tasks execute, MATLAB startup options in CI, and artifact collection for CI jobs.

For example, in your `.gitlab-ci.yml` file you can change the `script` field to specify different values for the `Tags`, `RerunFailedTasks`, and `PipelineArchitecture` properties in `padv.pipeline.GitLabOptions`:

```
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(
        Tags='high_memory',
        RerunFailedTasks = true,
        PipelineArchitecture = padv.pipeline.Architecture.SerialStages,
        GeneratedYMLFileName = 'simulink_pipeline.yml',
        GeneratedPipelineDirectory = fullfile('derived','pipeline')));
        "
```

This code specifies that the pipeline should be associated with the GitLab Runner tag `high_memory`, should try to rerun failed tasks, and should use a serial stage pipeline architecture that creates a job for each task iteration (for example, one job for running **Check Modeling Standards** on `ModelA` and one job for running **Check Modeling Standards** on `ModelB`). For more information about the available pipeline architectures, see the next section "Customize Pipeline Architecture".

To see a list of the available properties in the MATLAB Command Window, enter:

```
help padv.pipeline.GitLabOptions
```

Customize Pipeline Architecture

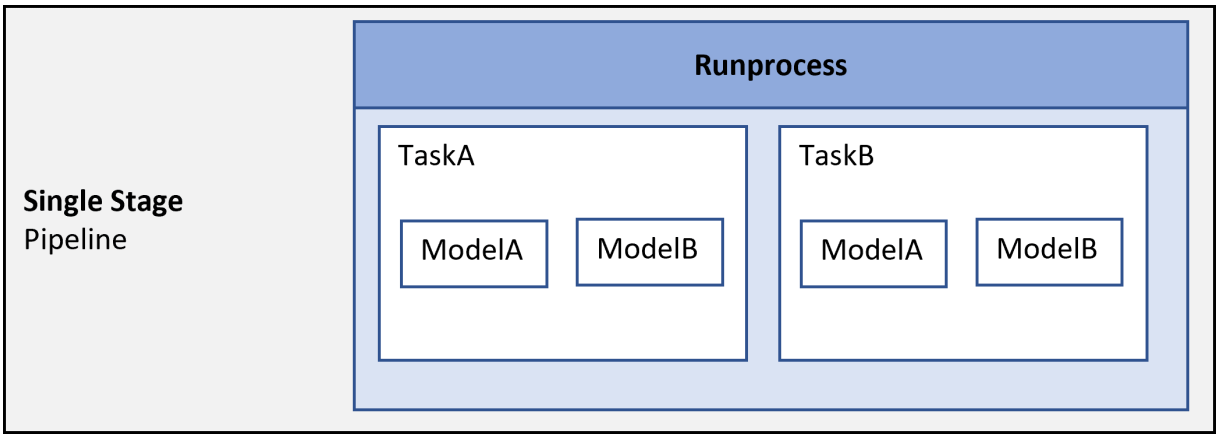
After you run a pipeline, GitLab shows the overall status of the pipeline and the status of each stage in the pipeline. For example, the **Stages** column can show a pipeline mini graph that shows the first stage passed, the second stage failed, and the third stage was skipped.

If you want to group the information that appears in your pipeline results, you can specify a pipeline architecture that defines more stages. If a pipeline has more stages, you can more easily identify where any failures occurred, but the pipeline execution might not be as efficient.

If you specify the pipeline architecture as:

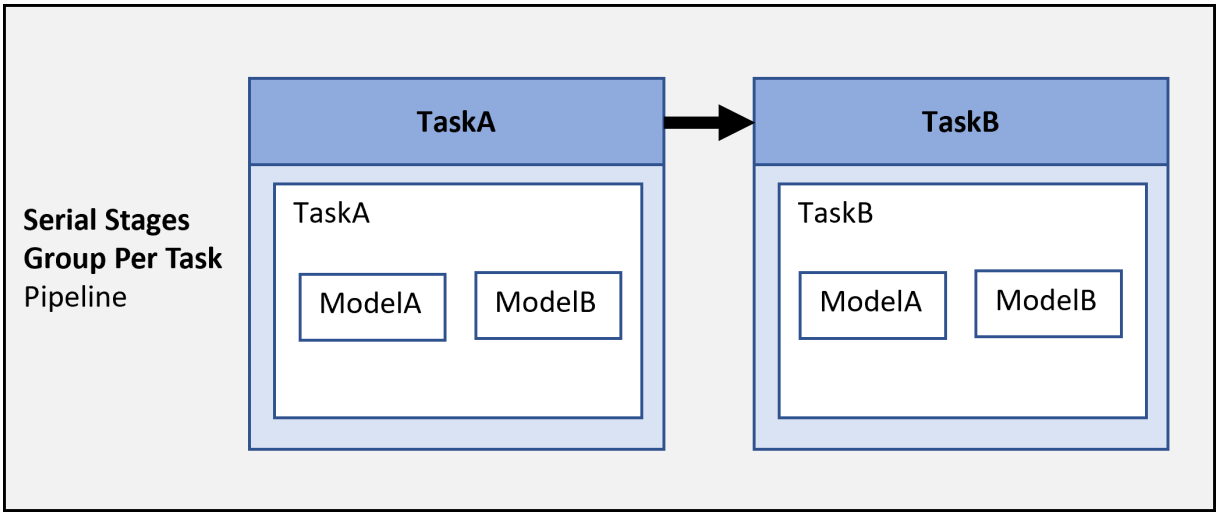
- `padv.pipeline.Architecture.SingleStage` — The generated pipeline contains a single stage, **Runprocess**, that runs all tasks.

  ```
  padv.pipeline.GitLabOptions(
  PipelineArchitecture = padv.pipeline.Architecture.SingleStage)
  ```



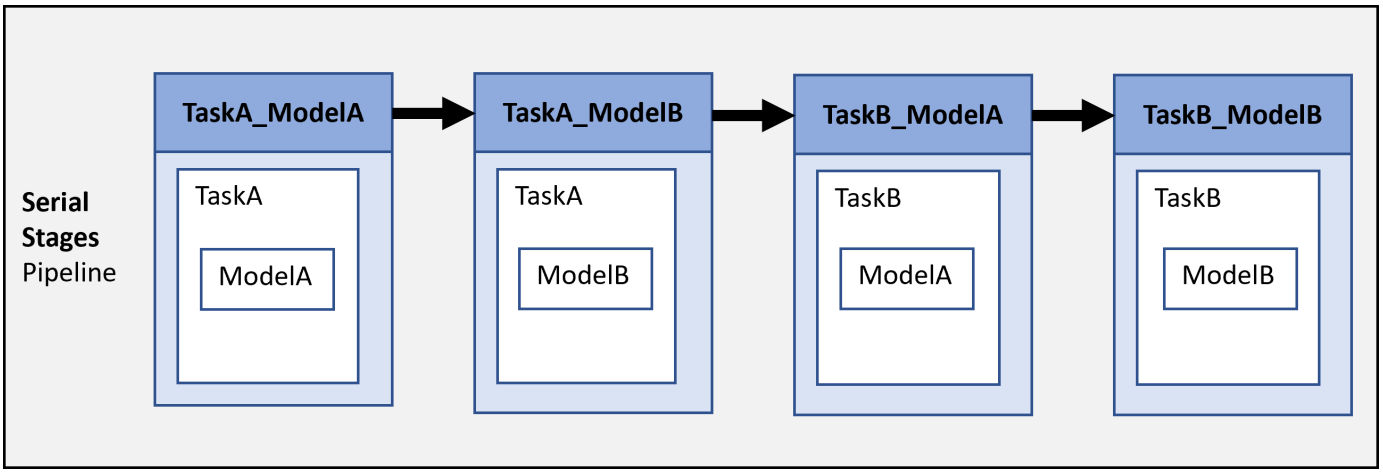
- `padv.pipeline.Architecture.SerialStagesGroupPerTask` — The generated pipeline contains one stage for each type of task.

  ```
  padv.pipeline.GitLabOptions(
  PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask)
  ```



- `padv.pipeline.Architecture.SerialStages` — The generated pipeline contains one stage for each task iteration.

  ```
  padv.pipeline.GitLabOptions(
  PipelineArchitecture = padv.pipeline.Architecture.SerialStages)
  ```

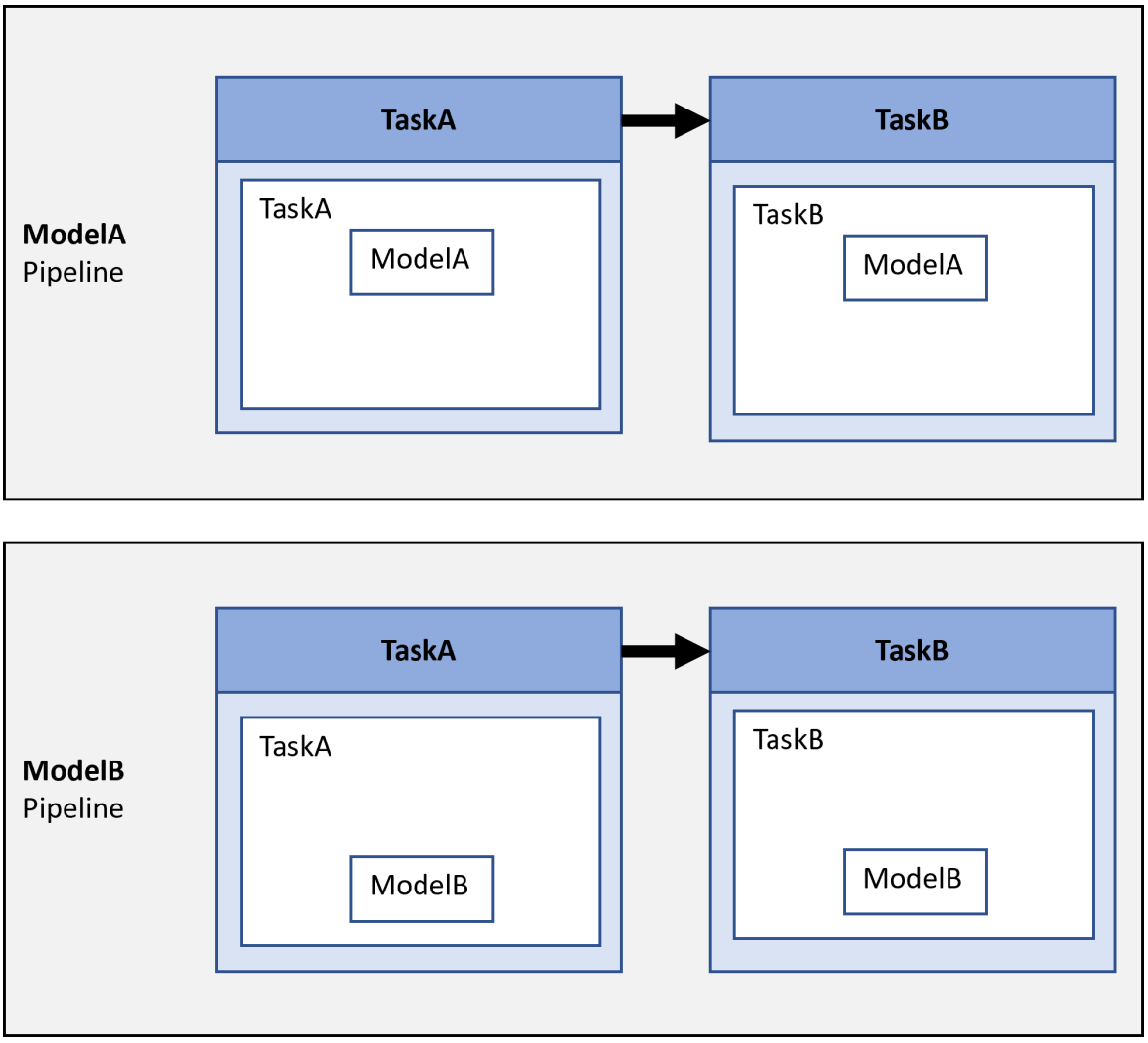


- `padv.pipeline.Architecture.IndependentModelPipelines` — The generated pipeline creates parallel pipelines, one for each model, that independently run the tasks associated with each model.

  ```
  padv.pipeline.GitLabOptions(
  PipelineArchitecture = padv.pipeline.Architecture.IndependentModelPipelines)
  ```

To make sure the jobs run in parallel, make sure that you either:

- Have multiple runners available. See https://docs.gitlab.com/ee/ci/yaml/#parallel.
- Configure your runner to run multiple jobs concurrently by specifying the `concurrent` setting. See https://docs.gitlab.com/runner/configuration/advanced-configuration.html.



Comparison of Pipeline Architectures

The following table compares the different pipeline architectures.

Type	Pipeline Architecture Value	Benefits	Limitations
Serial	`SingleStage`	One stage for all tasks. Efficient execution since the CI system only launches MATLAB and the project one time.	Difficult to identify where a failure occurred. If the pipeline fails, you must investigate the logs, build report, or other output files to identify which specified task or task iteration failed.

Type	Pipeline Architecture Value	Benefits	Limitations
	`SerialStagesGroupPerTask`	One stage for each task. The stages run in series, not in parallel. If the pipeline fails, you can see which task failed, directly in the pipeline results.	Less efficient execution because the CI system has to close and reopen MATLAB and the project one time for each stage
	`SerialStages`	One stage for each task iteration. The stages run in series, not in parallel. If the pipeline fails, you can see which task iteration failed, directly in the pipeline results.	Inefficient execution because the CI system has to close and reopen MATLAB and the project one time for each stage
Parallel	`IndependentModelPipelines`	Parallel pipelines, one for each model, independently run the tasks associated with each model. The pipeline executes efficiently because the tasks associated with each model run in parallel.	This pipeline architecture is only available for process models in which each model can run independently. If models depend on each other, you must use one of the serial pipeline architectures instead.

Integrate into Jenkins

A *pipeline* is a collection of automated procedures and tools that execute in a specific order to enable a streamlined software delivery process. CI systems allow you to define and configure a pipeline by using a pipeline file. In Jenkins, you can 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 CI/CD Automation for Simulink Check comes with an example Jenkinsfile that you can add to your project to run pipelines in Jenkins. When you use the example Jenkinsfile, the file generates and loads pipelines for you so that you do not need to manually update any pipeline files when you change the tasks and artifacts in your project.

Tip To see an example project that uses the example Jenkinsfile, open the Jenkins example project. In the MATLAB Command Window, enter:

```
processAdvisorJenkinsExampleStart
```

Integrate Using Default Options

Note This section assumes that Jenkins and your project are connected to your source control system. For an example of how to use GitLab for version control and Jenkins for continuous integration, see the Appendix in https://www.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models.html.

1. Connect your project to Jenkins by installing the following plugins on your Jenkins controller:
   - MATLAB Plugin for Jenkins. The plugin allows you to use the `runMATLABCommand` command to run MATLAB in freestyle and multi-configuration (matrix) Jenkins projects. For information, see the plugin on Jenkins Plugin Index: https://plugins.jenkins.io/matlab/
   - Jenkins Core Plugin, which allows pipelines to archive artifacts using the `archiveArtifacts` step. For information, see the Jenkins documentation: https://www.jenkins.io/doc/pipeline/steps/core/#archiveartifacts-archive-the-artifacts
   - JUnit Plugin, which allows Jenkins to show test failures and trends directly in the user interface. For information, see https://plugins.jenkins.io/junit/.
   - Job Cacher Plugin, which allows Jenkins to store caches. For information, see https://plugins.jenkins.io/jobcacher/.
2. Change your current folder to your project root and copy the example Jenkinsfile file into your project.

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

   copyfile(exampleJenkinsfile,"Jenkinsfile")
   ```

   Note The example Jenkinsfile is generic and can work with any project.

3. Open and inspect the `Jenkinsfile` file in your project.

   The file `Jenkinsfile` defines a parent pipeline. The parent pipeline uses the pipeline generator, `padv.pipeline.generatePipeline`, to automatically generate and execute an internal pipeline for your project. The options for the internal pipeline are specified by the object `padv.pipeline.JenkinsOptions`.

4. In your `Jenkinsfile`, update the file to use the:

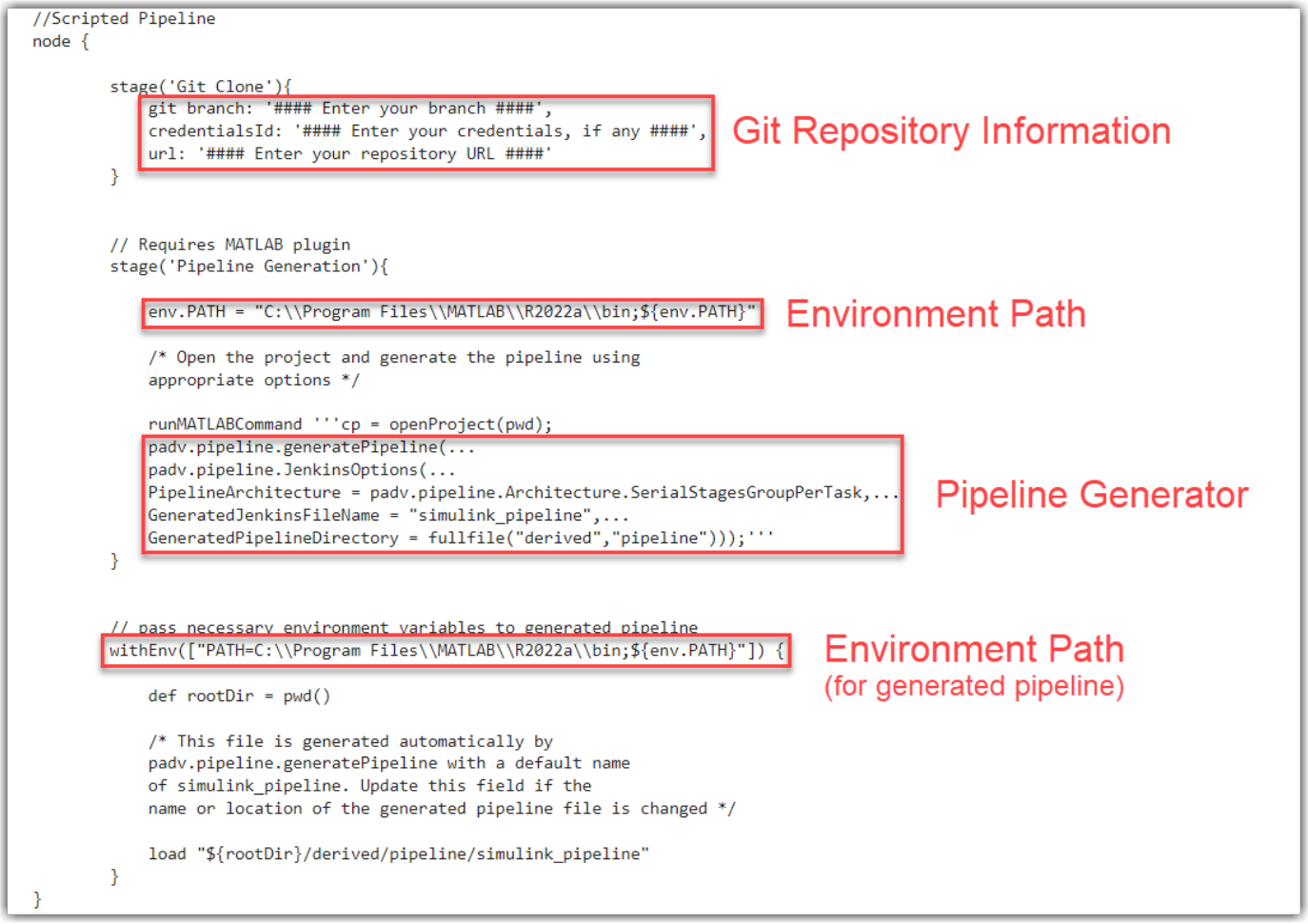
   - Git `branch`, `credentialsId`, and `url` for your repository. For example:

     ```
     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\\R2024a\\bin;${env.PATH}" // Windows`
       `// env.PATH = "/usr/local/MATLAB/R2024a/bin:${env.PATH}"        // Linux`
       `// env.PATH = "/Applications/MATLAB_R2024a.app/bin:${env.PATH}" // macOS`
     - `withEnv(["PATH=C:\\Program Files\\MATLAB\\R2024a\\bin;${env.PATH}"]) { // Windows`
       `// withEnv(["PATH=/usr/local/MATLAB/R2024a/bin:${env.PATH}"]) {        // Linux`
       `// withEnv(["PATH=/Applications/MATLAB_R2024a.app/bin:${env.PATH}"]) { // macOS`

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



5 Add your `Jenkinsfile` to your project.

6 Push the changes to your project to source control. If your Jenkins project is not automatically triggered by pushing changes to source control, manually trigger your Jenkins pipeline.

By default, a Jenkins project automatically considers the file `Jenkinsfile` at the root of the source control repository as the CI/CD configuration file for the build. Your Jenkins agent can now automatically generate and execute a custom, internal pipeline for your project each time a Jenkins build triggers.

Note You do not need to update the `Jenkinsfile` file if you make changes to your projects or process model. The pipeline generator generates the internal pipeline using the latest project and process model. You only need to update the `Jenkinsfile` file if you want to change how the pipeline generator organizes and executes the pipeline.

In Jenkins, your pipeline will contain two upstream jobs:

- **Git_Clone** — Loads your Git repository information.
- **Pipeline Generation** — Automatically generates and loads a downstream Jenkinsfile that defines a Jenkins pipeline for your process. By default, the downstream pipeline contains:

- One job for each task defined in the process model file
- One job, `Generate_PADV_Report`, that generates a Process Advisor build report
- One job, `Collect_Artifacts`, that collects build artifacts

The pipeline generator automatically generates JUnit-style XML reports for each task. Jenkins can use the JUnit reports to show test failures and trends directly in the user interface. For information on how Jenkins displays JUnit information, see the Jenkins documentation: https://plugins.jenkins.io/junit/. If you do not want to generate JUnit reports, specify the `GenerateJUnitForProcess` property in `padv.pipeline.JenkinsOptions` as `false`.

If you want to change how the downstream jobs get organized and executed, you can modify the properties of the `padv.pipeline.JenkinsOptions`. For example, you can modify the `PipelineArchitecture` property to change the number of stages and the grouping of tasks in each stage of the downstream pipeline.

For more information, see "Customize Downstream Pipeline" or enter this code in the MATLAB Command Window:

```
help padv.pipeline.JenkinsOptions
```

Customize Downstream Pipeline

You can use the properties of `padv.pipeline.JenkinsOptions` to control which Jenkins agent to associate with the downstream pipeline, the number of stages and the grouping of tasks in the downstream pipeline (defined by the pipeline architecture), how tasks execute, and artifact collection for CI jobs.

For example, in your `Jenkinsfile` file you can change the `Pipeline Generator` stage to specify different values for the `AgentLabel`, `RerunFailedTasks`, and `PipelineArchitecture` properties in `padv.pipeline.JenkinsOptions`:

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

    env.PATH = "C:\\Program Files\\MATLAB\\R2024a\\bin;${env.PATH}" // Windows
    // env.PATH = "/usr/local/MATLAB/R2024a/bin:${env.PATH}"        // Linux
    // env.PATH = "/Applications/MATLAB_R2024a.app/bin:${env.PATH}" // macOS

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

    runMATLABCommand '''cp = openProject(pwd);
    padv.pipeline.generatePipeline(...
    padv.pipeline.JenkinsOptions(...
    AgentLabel="high_memory",...
    RerunFailedTasks = true,...
    PipelineArchitecture = padv.pipeline.Architecture.SerialStages,...
    GeneratedJenkinsFileName = "simulink_pipeline",...
    GeneratedPipelineDirectory = fullfile('derived','pipeline')));'''
}
```

This code specifies that the pipeline should be associated with the Jenkins agent labeled `high_memory`, should try to rerun failed tasks, and should use a serial stage pipeline architecture that creates a job for each task iteration (for example, one job for running **Check Modeling**

Standards on `ModelA` and one job for running **Check Modeling Standards** on `ModelB`). For more information about the available pipeline architectures, see the next section "Customize Pipeline Architecture".

To see a list of the available properties in the MATLAB Command Window, enter:

```
help padv.pipeline.JenkinsOptions
```

Customize Pipeline Architecture

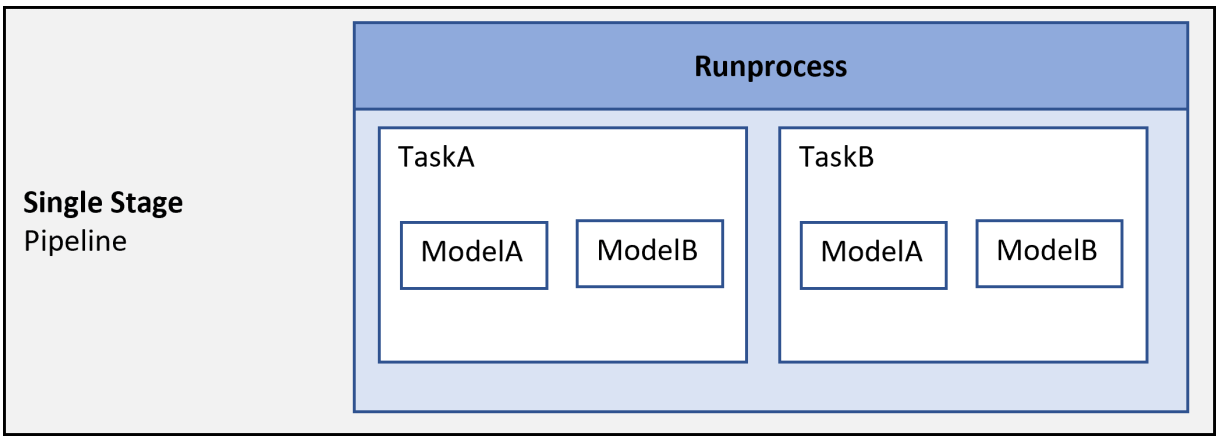
After you run a pipeline, the **Stage View** in Jenkins shows the status of each stage in the build.

To change the stages that appear in the **Stage View** for your automatically generated pipeline, you can specify a different pipeline architecture in the call to the pipeline generator. The pipeline architecture defines the number of stages in your pipeline and the grouping of tasks in each stage. If a pipeline has more stages, you can more easily identify where any failures occurred, but the pipeline execution might not be as efficient.

If you specify the pipeline architecture as:

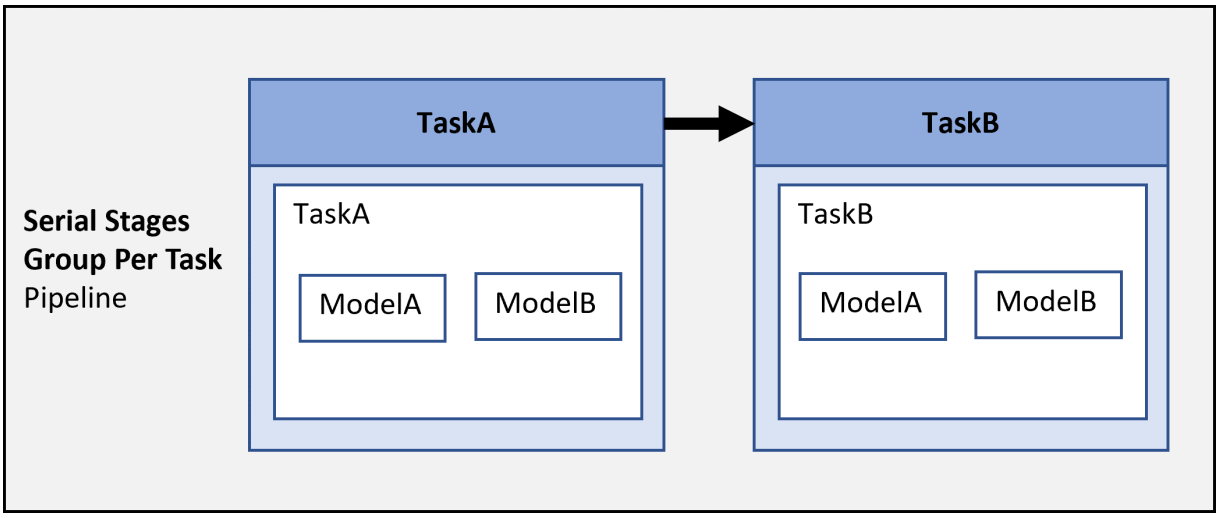
- `padv.pipeline.Architecture.SingleStage` — The generated pipeline contains a single stage, **Runprocess**, that runs all tasks.

  ```
  padv.pipeline.JenkinsOptions(...
  PipelineArchitecture = padv.pipeline.Architecture.SingleStage)
  ```



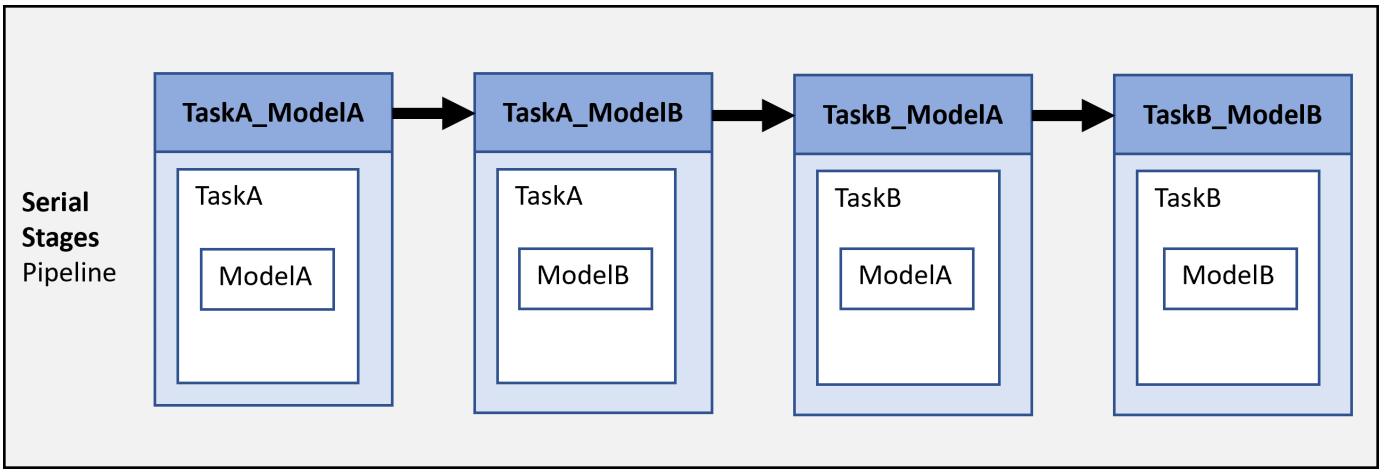
- `padv.pipeline.Architecture.SerialStagesGroupPerTask` — The generated pipeline contains one stage for each type of task.

  ```
  padv.pipeline.JenkinsOptions(...
  PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask)
  ```



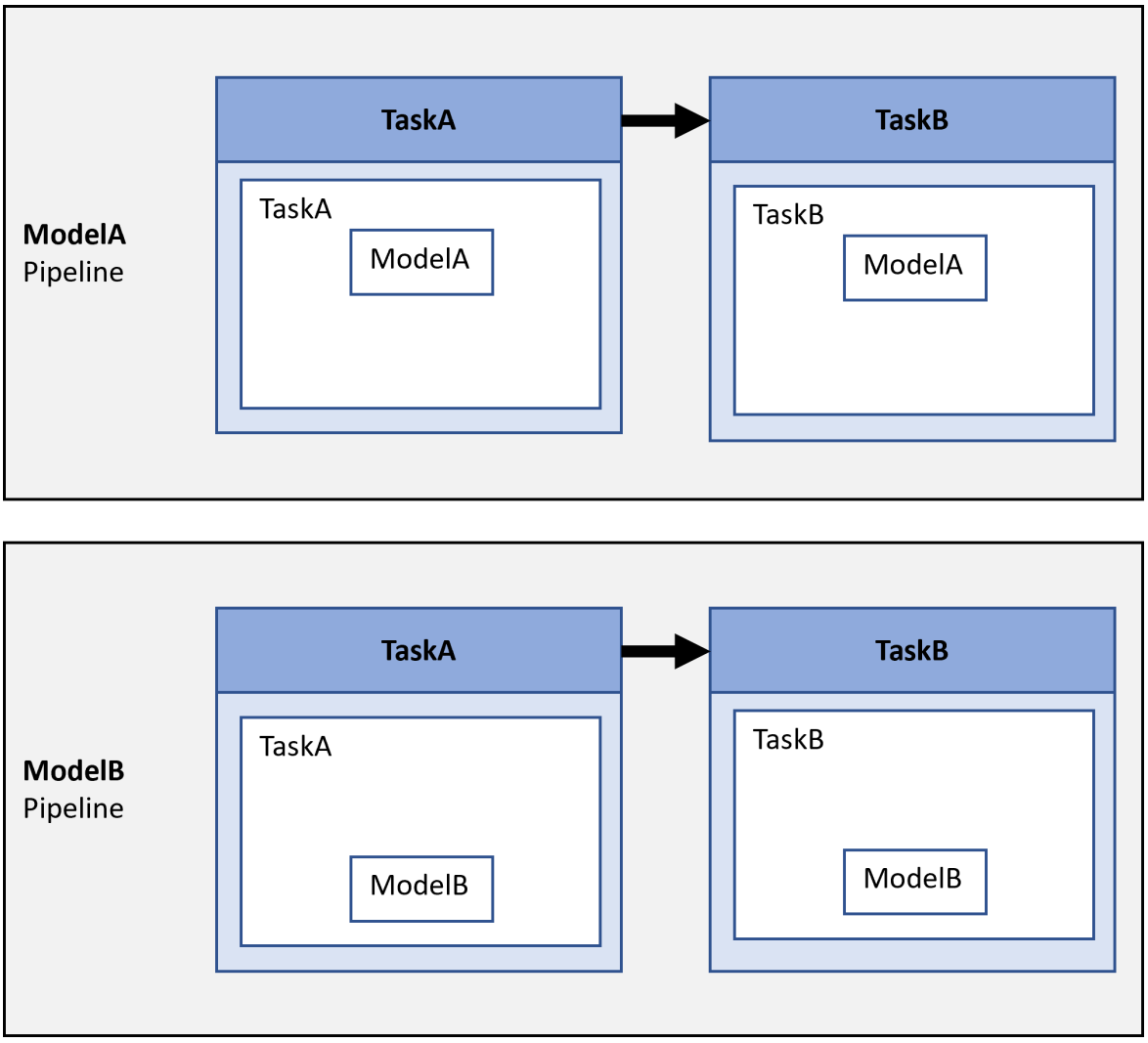
- `padv.pipeline.Architecture.SerialStages` — The generated pipeline contains one stage for each task iteration.

```
padv.pipeline.JenkinsOptions(...
PipelineArchitecture = padv.pipeline.Architecture.SerialStages)
```



- `padv.pipeline.Architecture.IndependentModelPipelines` — The generated pipeline creates parallel pipelines, one for each model, that independently run the tasks associated with each model.

```
padv.pipeline.JenkinsOptions(
PipelineArchitecture = padv.pipeline.Architecture.IndependentModelPipelines)
```



Comparison of Pipeline Architectures

The following table compares the different pipeline architectures.

Type	Pipeline Architecture Value	Benefits	Limitations
Serial	`SingleStage`	One stage for all tasks. Efficient execution since the CI system only launches MATLAB and the project one time.	Difficult to identify where a failure occurred. If the pipeline fails, you must investigate the logs, build report, or other output files to identify which specified task or task iteration failed.

Type	Pipeline Architecture Value	Benefits	Limitations
	`SerialStagesGroupPerTask`	One stage for each task. The stages run in series, not in parallel. If the pipeline fails, you can see which task failed, directly in the **Stage View**.	Less efficient execution because the CI system has to close and reopen MATLAB and the project one time for each stage
	`SerialStages`	One stage for each task iteration. The stages run in series, not in parallel. If the pipeline fails, you can see which task iteration failed, directly in the **Stage View**.	Inefficient execution because the CI system has to close and reopen MATLAB and the project one time for each stage
Parallel	`IndependentModelPipelines`	Parallel pipelines, one for each model, independently run the tasks associated with each model. The pipeline executes efficiently because the tasks associated with each model run in parallel.	This pipeline architecture is only available for process models in which each model can run independently. If models depend on each other, you must use one of the serial pipeline architectures instead.

Integrate into Other CI Platforms

You can use any of the MATLAB-supported continuous integration (CI) platforms to run your automated pipeline of tasks. For information on the supported platforms, see https://www.mathworks.com/help/matlab/matlab_prog/continuous-integration-with-matlab-on-ci-platforms.html.

Run MATLAB in Batch Mode

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, `matlab -batch "runprocess"` starts MATLAB noninteractively and runs each of the tasks in the pipeline defined by the process model file (`processmodel.p` or `processmodel.m`) in the project. MATLAB terminates automatically with the exit code `0` if the specified code executes successfully without error. Otherwise, MATLAB terminates with a nonzero exit code.

Generate and Run Pipeline Using runprocess Function

You can generate and run your pipeline in CI by using the `runprocess` function. By default, the `runprocess` functions runs all the tasks in your process, but you can also use one or more name-value arguments to specify how the pipeline runs. For example:

Run All Tasks

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

```
runprocess()
```

Run Specific Task

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 Tasks for Specific Artifact

To only run the tasks associated with a specific artifact, use the `FilterArtifact` argument. For example, to only 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 "runprocess" in the Reference Book PDF.

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.

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. For more information, see "Set Up Virtual Display for No-Display Machine".

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. 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"
   -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. For example:

   ```
   docker run --init --rm my_matlab_image_name -batch "ver"
   ```

   For information, see https://docs.docker.com/reference/cli/docker/container/run/

Troubleshooting and Limitations

Troubleshooting Missing Tasks or Artifacts

When you use CI/CD Automation for Simulink Check, the support package creates a digital thread to capture the attributes and unique identifiers of the artifacts in your project. The digital thread is a set of metadata information about the artifacts in a project, the artifact structure, 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.

See the next sections for troubleshooting steps and limitations.

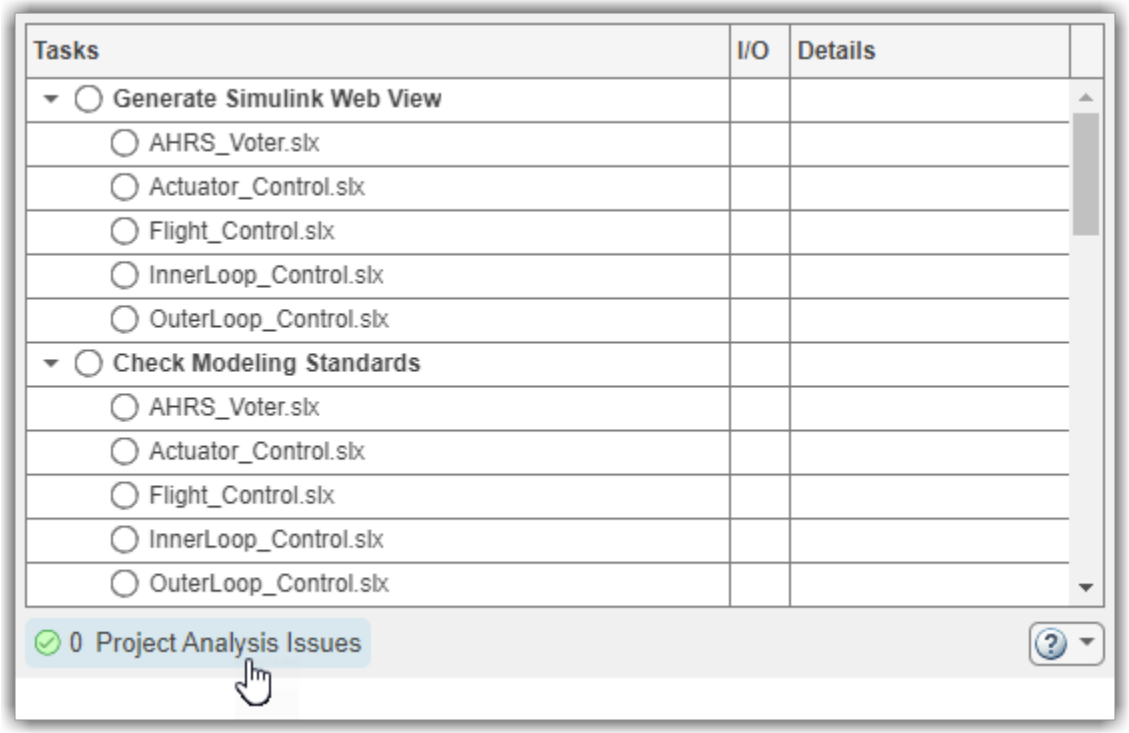
Artifact Issues

Before you begin troubleshooting Process Advisor or the build system:

- Check that artifacts are saved in the project.
- If you are using R2022b, check that artifacts are not in a referenced project. Project references are supported starting in R2023a.
- 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 any errors or warnings that were generated during artifact analysis.



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 artifact, 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.

If there are issues with an artifact, check that the artifact does not use the following unsupported modeling constructs:

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
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 https://www.mathworks.com/help/ecoder/ug/library-based-code-generation-for-subsystems-shared-across-models.html.

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.

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

Note 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");
```

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 *untracked dependency*.

The table in this section lists the known untracked dependencies.

- The **Artifact** column lists the artifacts with known untracked dependencies.
- The **Untracked 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.

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 shows:

- **Artifact**: Model
- **Untracked Dependency**: Referenced global workspace variable

Artifact	Untracked 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: • 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. The digital thread tracks changes to data dictionaries.

Note 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 the section "How to Disable Incremental Builds".

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`.

Note The build system and Process Advisor app are able to track the following test case dependencies:

- Baseline files in `.mat`, `.xlsm`, `.xlsb`, `.xlsx`, `.xls`, and `.mldatx` format.
- Input files in `.mat`, `.xlsm`, `.xlsb`, `.xlsx`, and `.xls` format.
- Parameter override files in `.mat`, `.xlsm`, `.xlsb`, `.xlsx`, `.xls`, and `.m` format.

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>`
- If a test harness is saved inside a model file, the Process Advisor and build system return an incorrect warning that the internal test harness is not on the MATLAB search path. Ignore the warning, and, if possible, convert your internal test harnesses to external test harnesses so that the support package can differentiate between changes to the test harness and changes to the main model.
- When you add the built-in task **Check Coding Standards** (`padv.builtin.task.AnalyzeModelCode`) to your process model, you must add code that checks if Polyspace is installed and setup. Otherwise, you get an error message: `Unrecognized function or variable 'polyspaceroot'.`

  For more information on this task, see "Check Coding Standards or Prove Code Quality" in the Reference Book PDF.
- 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 to Polyspace Access" in the Reference Book PDF.

  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;`.

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.

Note In R2022b, if a test harness is saved inside a model file, the Process Advisor and build system return an incorrect warning that the internal test harness is not on the MATLAB search path. Ignore the warning, and, if possible, convert your internal test harnesses to external test harnesses so that the support package can differentiate between changes to the test harness and changes to the main model.

To convert a test harness, open Simulink Test for the main model and, on the **Tests** tab, click **Manage Test Harnesses > Convert to External Harnesses**. Click **Yes** to convert the affected test harnesses.

Set Up Virtual Display for No-Display Machine

Issue

Some MATLAB code, including some built-in tasks, can only run successfully if a display is available for your machine. If 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.

Workaround

As a 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.

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 `padv.util.forceReanalyzeProject` in the Reference Book PDF.

Note You should only use the function `padv.util.forceReanalyzeProject` if 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` in the Reference Book PDF.

Version History

April 2024

Released for:

- 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 `IndependentModelPipeline` 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".

- 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 into GitHub" or "Integrate into GitLab".

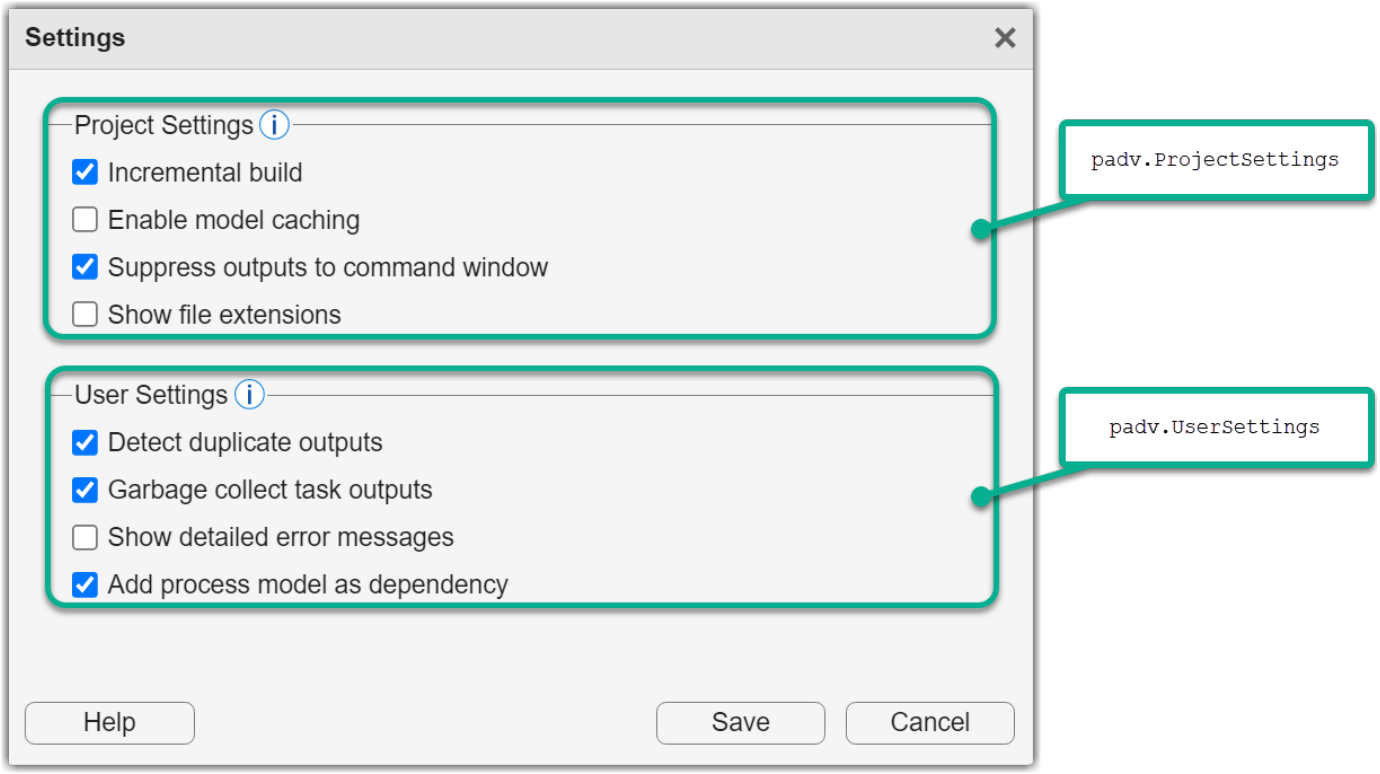
- Previously, if you wanted to create a Docker image that installed the support package, you needed to download and use the offline installer files. You can now build a Docker image that directly installs the support package and other products using the MATLAB Package Manager (MPM). For information and the updated example Dockerfile, see "Create Docker Container for Support Package".

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.

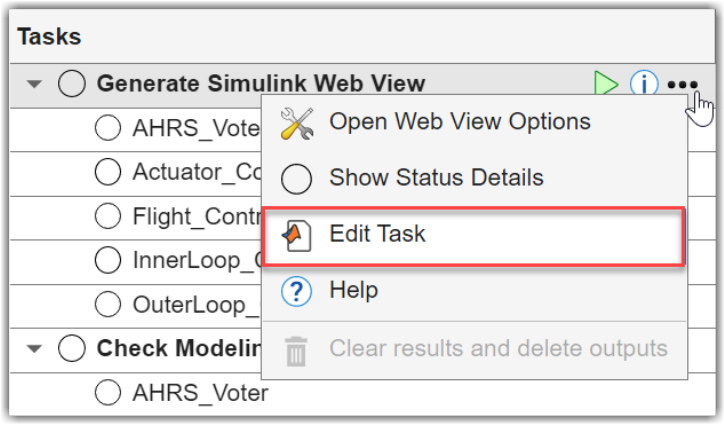
  Additionally, you no longer need to create a project startup script to persist run-time settings. The `padv.UserSettings` class automatically manages and persists those settings across MATLAB sessions on your machine.

  The main class properties correspond to settings in the Process Advisor Settings dialog box.



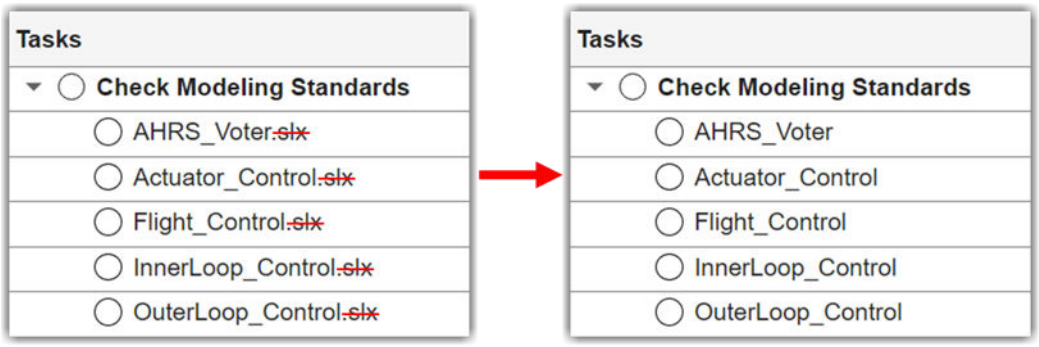
The class `padv.Preferences` will be removed in a future release. For project settings, use `padv.ProjectSettings` instead. For run-time settings, use `padv.UserSettings` instead.

- 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 "Author Your Process Model".

- 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.



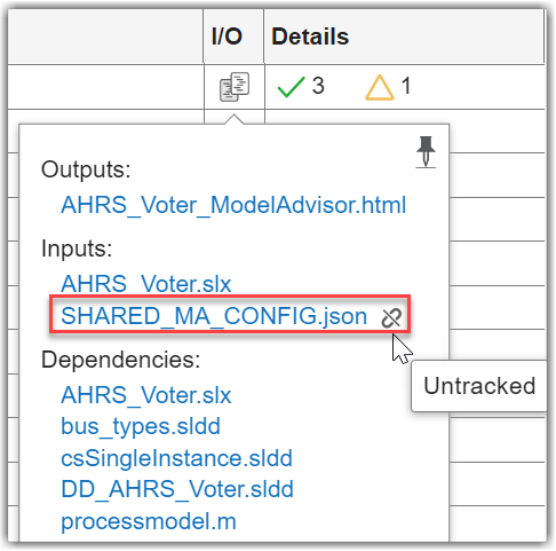
By default, queries now strip file extensions from the `Alias` property of each task iteration artifact. To show file extensions for all artifacts in the **Tasks** column, select the project setting **Show file extensions**. To keep file extensions in the results for a specific query, specify the query 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')));
```

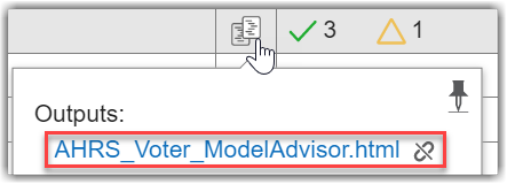
  In the Process Advisor **I/O** column, the file appears as **Untracked** because you cannot track changes to files outside the project. If you make a change to an untracked file, the build system does not mark the task as outdated. For more information, see "Turn Off Change Tracking for Input Artifacts".



- 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. For more information, see "Turn Off Change Tracking for Task Outputs".



- The build system can now cache requirement sets. For information on caching, see "Cache Models and Other Artifacts Used During Build" in this PDF.

- 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" in the Reference Book PDF.

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 "Check Coding Standards or Prove Code Quality" in the Reference Book PDF.

- 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 "padv.builtin.query.FindCodeForModel" in the Reference Book PDF.

- 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` in the Reference Book PDF.

- 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` in the Reference Book PDF.

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 `padv.Preferences` with `padv.UserSettings.get()`.
`GarbageCollectTaskOutputs`	
`ShowDetailedErrorMessages`	
`TrackProcessModel`	
`FilteredDigitalThreadMessages`	Replace instances of `padv.Preferences` with `padv.ProjectSettings.get()`.
`IncrementalBuild`	
`EnableModelCaching`	
`MaxNumModelsInCache`	
`MaxNumTestResultsInCache`	
`SuppressOutputWhenInteractive`	

  For example:

Functionality	Use This Instead

`% changing run-time setting`
`p1 = padv.Preferences;`
`p1.DetectDuplicateOutputs = false;`
	`p1 = padv.UserSettings.get();`
	`p1.DetectDuplicateOutputs = false;`

`% changing project setting`
`p1 = padv.Preferences;`
`p1.IncrementalBuild = false;`
	`p1 = padv.ProjectSettings.get();`
	`p1.IncrementalBuild = false;`

- 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).[1]

Features

Model and Simulation Management

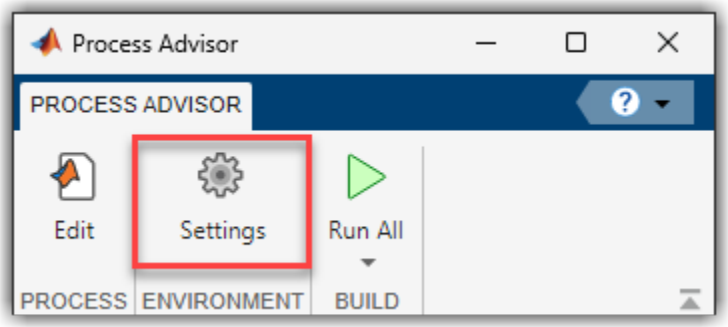
- Starting in R2023a, you can use your Model Advisor justifications when checking modeling standards. Provide your justification files as inputs to the task by using the new built-in query `padv.builtin.query.FindMAJustificationFileForModel` to find the justification files in a specified folder. For example:

```
maTask = addTask(pm,padv.builtin.task.RunModelStandards);
maTask.addInputQueries(...
    padv.builtin.query.FindMAJustificationFileForModel(...
    JustificationFolder=fullfile("Justifications","ModelAdvisor")));
```

  See `padv.builtin.query.FindMAJustificationFileForModel` in the Reference Book PDF.
- 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. For an example, see "Create Multiple Instances of Tasks".
- The property `DefaultOutputDirectory` for `padv.ProcessModel` now supports paths relative to the project root.

Process Advisor Enhancements

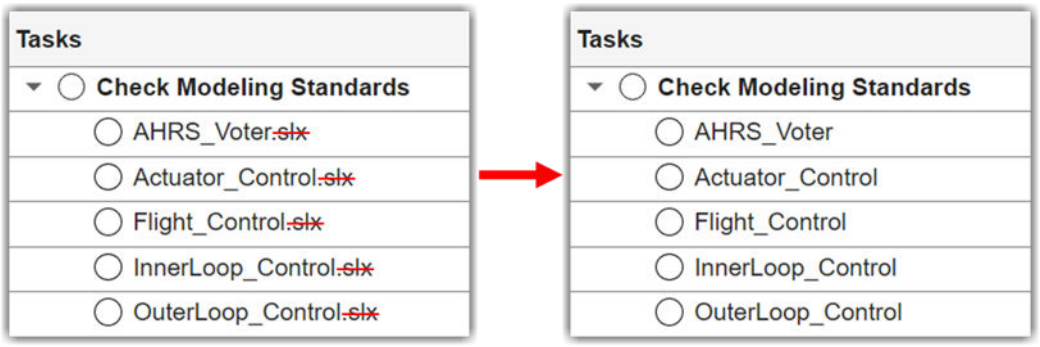
- You can now specify `padv.Preferences` by using the new **Settings** user interface in the Process Advisor. See "Specify Settings for Builds".



(continues on next page)

- You can now customize how artifact names appear in Process Advisor by using the new `Alias` property of `padv.Artifact` objects. For an example, see "Hide File Extension in Process Advisor" in this PDF.

1 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 Builds" in this PDF.

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" in this PDF.

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` in the Reference Book PDF.

Compatibility Considerations

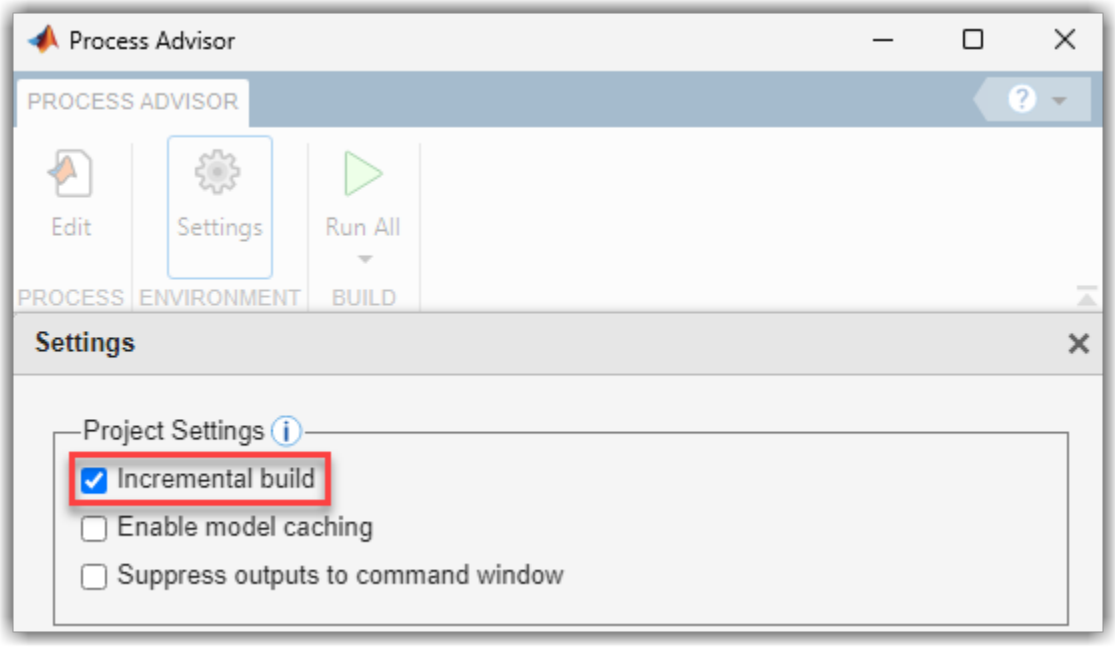
- **Supported Releases**
  - The February release is the last planned release for R2022a.

    In March 2024, the support package will support:

    - R2023b
    - R2023a
    - R2022b Update 1 (and later updates)

(continues on next page)

- **Process Advisor**
  - In Process Advisor, the **Incremental Build** check box is now in the Settings dialog box. In the toolstrip, 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` in the Reference Book PDF.

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 "Merge Test Results" in the Reference Book PDF.

- 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" in the Reference Book PDF.

- 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" in the Reference Book PDF.

- 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` in the Reference Book PDF.

(continues on next page)

⚠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 the process model. 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 "Check Coding Standards or Prove Code Quality" in the Reference Book PDF.

- 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 "padv.builtin.query.FindFileWithAddress" in the Reference Book PDF.

(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

`padv.builtin.query.FindFileWithAddress(...`
`"`*`artifactType`*`",...`
`"`*`filePath`*`")`
	`padv.builtin.query.FindFileWithAddress(...`
	`Type = "`*`artifactType`*`",...`
	`Path = "`*`filePath`*`")`

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`.

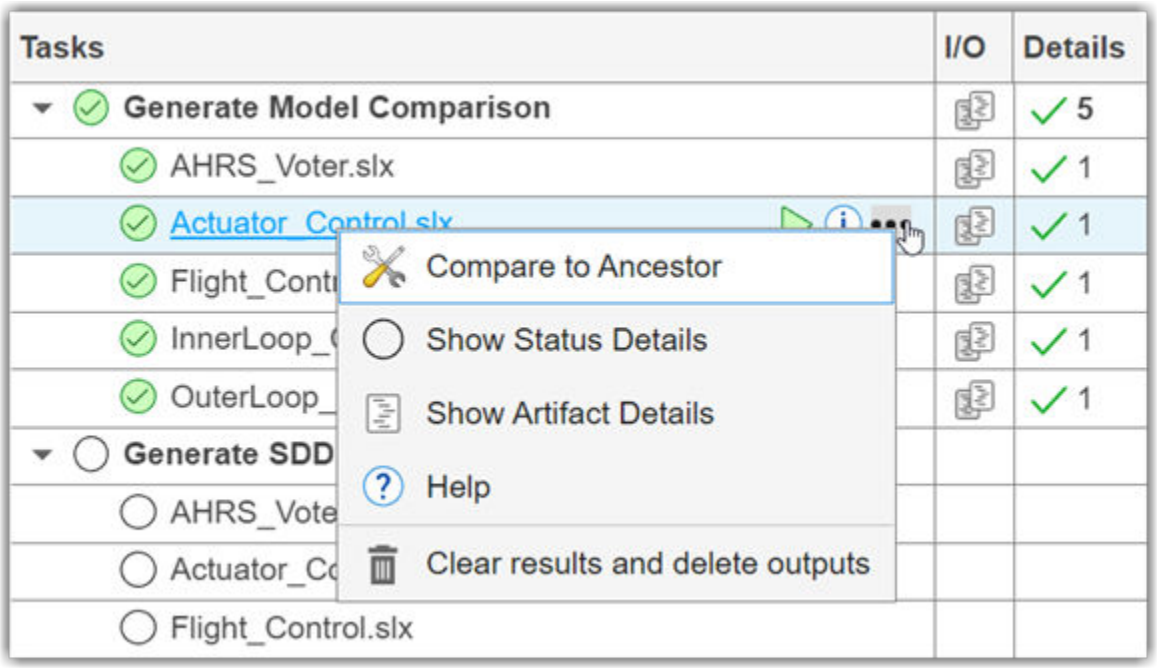
  To add the task to your process model, use the function `addTask`:

  ```
  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.



  For more information, see "Generate Model Comparison" in the Reference Book PDF.

(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 for No-Display Machine".

- 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

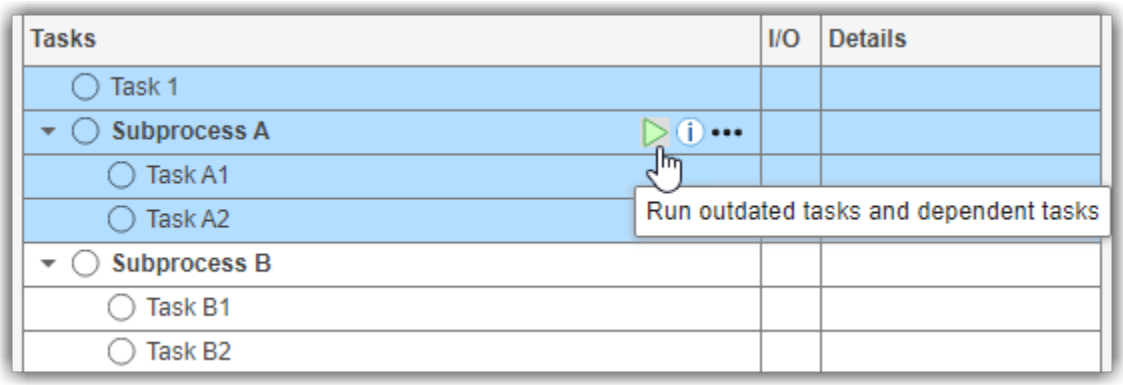
Supports:

- 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 into GitHub".

- 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".



- 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")
    ```

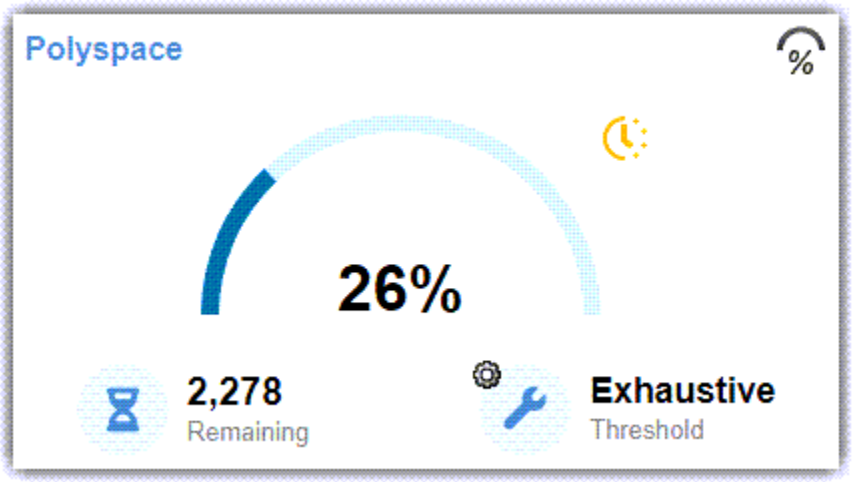
  - `FilterArtifact` — Specify the artifact that you want to run tasks on.

    ```
    runprocess(...
    FilterArtifact = fullfile("02_Models","AHRS_Voter","specification","AHRS_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" in the Reference Book PDF.

(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 "Check Coding Standards or Prove Code Quality" in the Reference Book PDF.

- By default, a query can find any artifact 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","AHRS_Voter","specification","AHRS_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 Automatic Pipeline Generation Works".

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: [0×0 padv.Artifact]
       ArtifactAddress: [1×1 padv.util.ArtifactAddress]
    ```

    For information, see "padv.util.ArtifactAddress" in the Reference Book PDF.
- **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 into GitLab" or "Integrate into Jenkins".
- 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".
- `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. For more information, see the Reference Book PDF.
- 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
`padv.builtin.task.AnalyzeRefModelCode`	`padv.builtin.task.AnalyzeModelCode`
`padv.builtin.task.AnalyzeTopModelCode`	
`padv.builtin.task.GenerateCodeAsRefModel`	`padv.builtin.task.GenerateCode`
`padv.builtin.task.GenerateCodeAsTopModel`	
`padv.builtin.task.RunCodeInspectionAsRefModel`	`padv.builtin.task.RunCodeInspection`
`padv.builtin.task.RunCodeInspectionAsTopModel`	

(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
`maTask.RunOptions.ReportPath`	`maTask.ReportPath`

  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
```

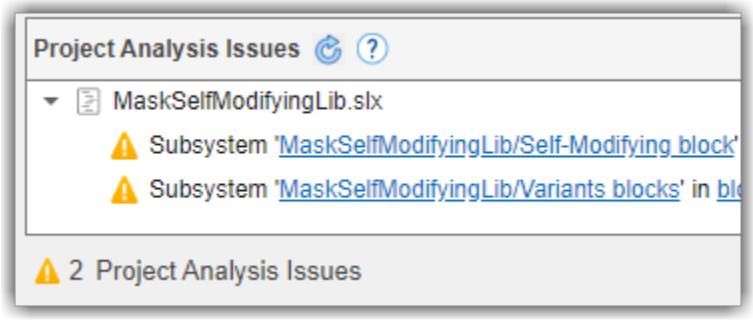
March 2023

Supports:

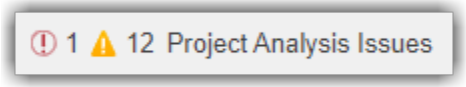
- R2023a
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

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.



For more information, see "Quick Reference for 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 the section "Integrate into Jenkins".

- 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 any 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:

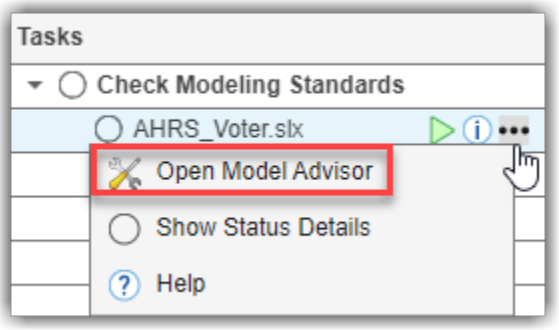
- R2022b Update 1 (and later updates)
- R2022a Update 4 (and later updates)

Features:

- Automatically generate a pipeline configuration file for a GitLab pipeline by using the new function `padv.pipeline.generatePipeline`. For more information, see the section "Integrate into GitLab" 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.
- If 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. For more information, see the section "How to Disable Incremental Builds".
- 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)