Search documentation Search

Appian

# Creating Memory Efficient Models Best Practices

Appian 7.7

**NOTE:** This site does not include documentation on the latest release of Appian. Please upgrade to benefit from our newest features. For more information on the latest release of Appian, please see the Appian 7.8 documentation

The information on this page is provided by the Appian Center of Excellence

Toggle Na

This page is *not* supported by Appian Technical Support. For additional assistance, please contact your Appian Account Executive to engage with Appian Professional Services.

Design decisions early in development can have long term effects on memory management. There are certain factors that a developer should take into account because they will affect the memory footprint of a process.

The first factor to consider is the size of the process model itself. Process models with a large amount of nodes will have a large memory footprint. The next aspect to take into account is the size of the data stored in process variables. The greater the number of process variables and the larger the amount of data stored in process variables, the more memory a process instance will consume. Next, process models that employ looping will generate large amount of process history which will have an impact on memory. Lastly, the number of process instances in memory at the same time will lead to greater process memory usage.

The general best practice for process design is to have short lived processes, which will minimize the impact of that particular process on engine memory size. Short lived processes allow parts of the process to be archived sooner than other parts which may need to be retained for longer. Also, short lived processes are easier to maintain and evolve. When a process model is upgraded, new process instances will use the new process model and existing process instances do not get updated. Shorter lived processes will result in the faster distribution of updated process models to end users.

This page explains how various business requirements can be satisfied using short lived processes as well as how to minimize memory used by long-lived processes.

## Design Patterns

### Record Centric Applications

Many applications are case-management applications in which the user finds a record and performs ad-hoc actions on it. This is ideally suited to data being stored in a database (entity-backed) or obtained from an external service (service-backed). This allows short lived processes to be started and completed quickly without the need for long lived orchestration processes. Record centric applications represent the most important data as Records. The record should be entity-backed or service-backed and it should expose related actions for users to perform work on the data.

In the case where not all work can be driven from related actions, it is still possible to have an underlying process model that manages the scheduling of tasks or events for the record. This underlying, long-lived process model should have as few nodes and process variables as possible to reduce memory use and all work (whether attended or unattend) should be delegated to sub-processes. Inside the sub-processes, data needed for tasks or events should be retrieved from the database or service just before the data is needed and written back to the database as soon as possible. In this way, the short-lived process will use very little memory.

### Modular Design with Sub-processes

Look for opportunities to break apart long processes into sub-processes. This reduces memory use because sub-process can be archived as soon as they are completed.

Often times long-lived processes can be broken apart by milestone, phase, or state. If a model remains active because it contains a timer or rule event that waits for a long time, consider instead terminating the process and replacing the timer or rule event with another process that runs periodically or polls a database to determine when to start the next phase of the process.

### Handling Asynchronous Integrations

A common anti-pattern is to design a long-lived process to call an external system and then wait a long time for a response from the external system. Rather than waiting for the asynchronous integration to respond, the process should terminate after calling the external system. Expose a separate process that can be initiated by the external system, either using a web-service call or message, in order to start the next phase of the overall workflow.

Alternatively, a separate process can be created to periodically poll the asynchronous endpoint until a response is received, archiving as soon as it's finished polling. Design this polling process to handle all responses from the asynchronous integration. An anti pattern would be to have 1000s of separate polling process listening for different responses on the same endpoint. That asynchronous process would then initiate the next phase in the overall workflow.

# Other Design Concerns

- Use activity class parameters (as opposed to process variables) when possible to limit the process history size. Activity class parameters do not affect process memory usage unless the form is specifically configured to be kept in memory.
- Limit the number of nodes and process variables in a process model by performing transformations on node outputs using activity class parameters instead of storing temporary data in process variables and later performing the transformation in a separate script task.
- Release the system memory used by completed nodes by checking the Delete previously completed/cancelled instances setting.
- Avoid retrieving unnecessary data from the database by using `queryEntity()` to retrieve only the needed data. Only use `queryRecord()` or query rules when `queryEntity()` isn't sufficient to perform the necessary external queries. See Querying Data from an RDBMS for a comparison of `queryRecord()`, `queryEntity()` and query rules.
- Only pull data that is necessary from external systems to perform the given calculation. Use PagingInfo to limit the amount of data that is pulled back from the external system. Not only will this minimize the overhead of storing a larger amount of data in the process variable on the process model but it will also make queries run faster. This applies for any function that takes PagingInfo as a parameter (e.g., `getPortalReportDatasubset()` `queryEntity()` or query rules).

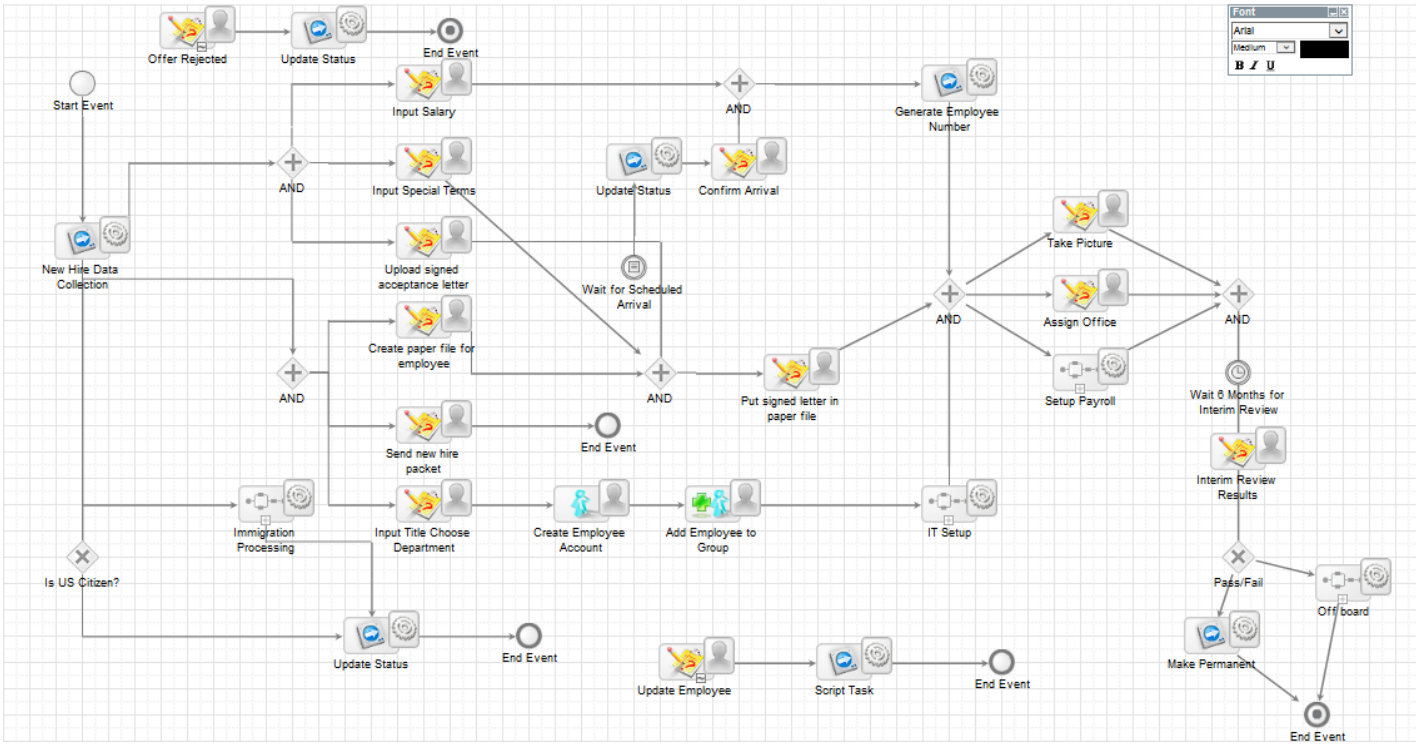
# Validating Memory Efficient Designs

The Process Sizing Script is one tool which will allow you to validate a memory efficient design. The process sizing script will identify the longest running processes as well as the processes that are taking up the most space.

# Example Walkthrough

## Employee Onboarding App

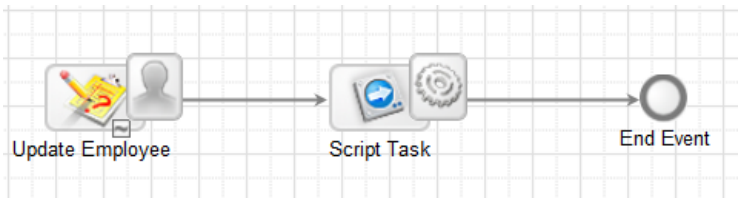
Let's examine a process model that wasn't created using a memory efficient design. The process model is an HR Onboarding Application. The HR Onboarding Application below resides in memory on average for 9 months. The HR Onboarding application includes making an offer to an employe, waiting for their arrival and again waiting for their interim review. Variables are stored in process and not in the database. Any updates to process variabl must occur through quick tasks.



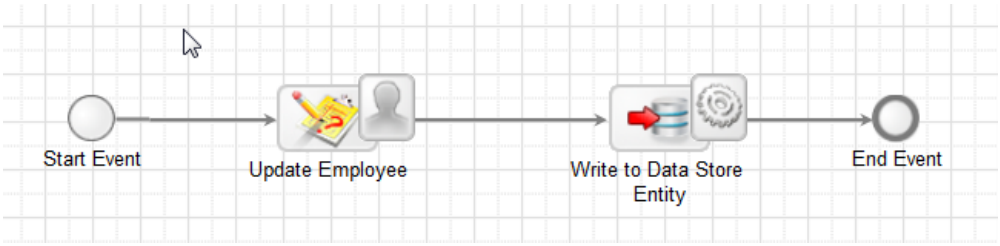

There are a couple of ways to improve the above process model. The first step is to identify what will be the central user record. That way changes occ on the record and not on in memory process variables through quick tasks. The central record in this case will be the user record.

The next step is to convert all ad-hoc quick tasks into standalone process model related actions.

Old:

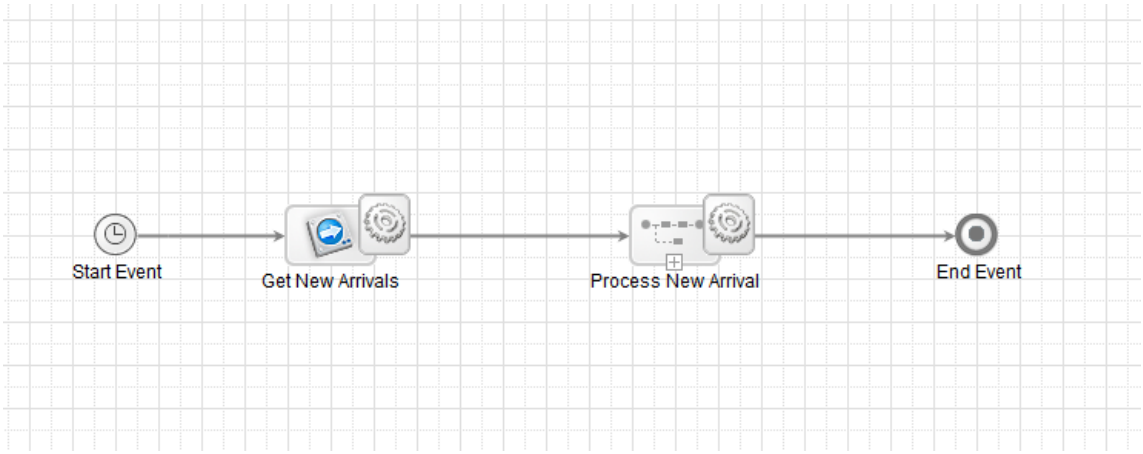



New:

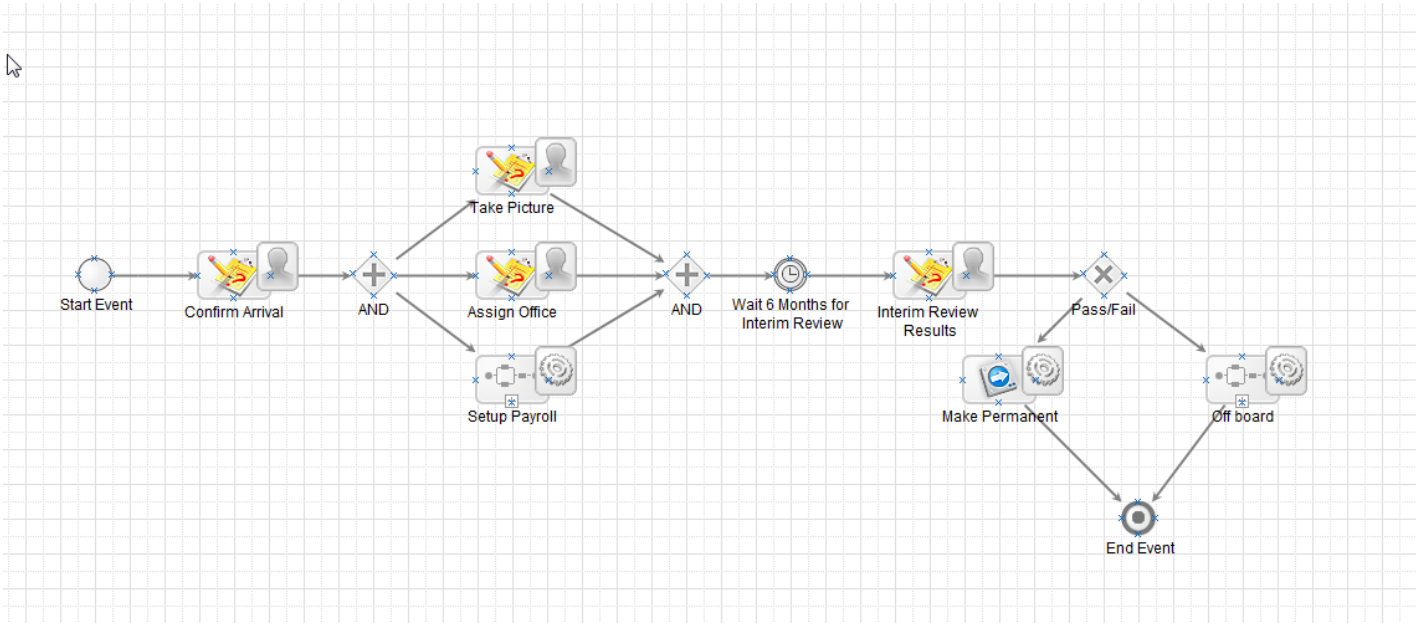



In the example above, the quick task was moved to its own process model. The Update Employee now retrieves the latest information from the databa and writes the user entered changes back to the datastore. Instead of going to the process dashboard, users now go to the user record for the latest information on an employee.

The next step is to break up the process into separate sub-processes by milestone. In this example, a series of task are completed and then the proces waits for the employee's arrival. A better solution would be to end the process once the pre-arrival tasks are complete. The portion of the process that waits for the employee's arrival could then resume once the employee arrives.




Note that in the example above we have a process that on a nightly basis that gets all of the new arrivals for the next morning. Next, the process launches a sub-process (shown below) to process the new arrival.




Note that the model above is the last half of the big process model that was shown in the first example. The key difference here is that over half of the model is now archived. Whereas, before the whole model was kept in memory. There is still another opportunity to further break this model up. Notice how there is a timer to wait 6 months for an interim review Instead of having the model wait for the review, there can be another nightly process which triggers the interim review six months after the date of hire.

User Guides by Role

- Designer
- Developer
- Web Admin
- Server Admin (On-Premise Only)

Tutorials

- Records
- Interfaces
- Process

Release Information

- Release Notes
- Installation
- Migration
- System Requirements
- Hotfixes
- Release History

Other

- STAR Methodology
- Best Practices
- Glossary
- APIs