Low-code Best Practices

for Profound.js and Profound API

August 12, 2024

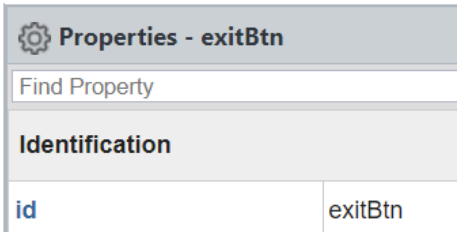
Contents

Overview

Best practices for low-code routines for Profound.js and Profound API.

Profound.js Rich Display Files

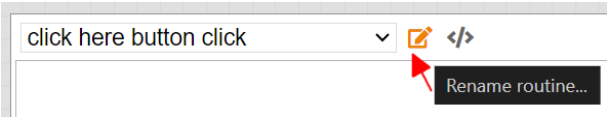
- Modularize Low Code
  Keep each RDF simple – do not put an excessive amount of logic or screens into a single RDF, this makes maintenance and debugging more difficult.
  - **Example:** Order Header and Order Detail Screens should be placed into separate RDF files.
  - Adopt IBMi "Work Panel" Approach (Dashboards)
    - Work with Grids
    - Display/Update/Add
- *NOTE: The more diligent you are in naming low code routines, elements, program fields; the easier maintenance becomes.
- After adding the widget to the screen, give it a descriptive ID as low code will refer to this as the UI element name (for client-side JavaScript).



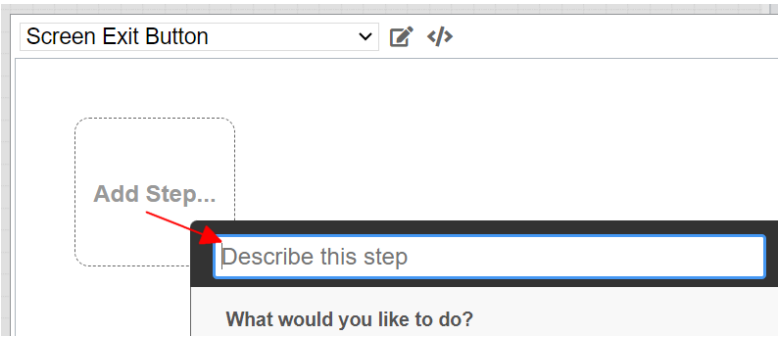
Low-code routines

Creating Routines and Steps

- Give the routine a descriptive name. After clicking Build Logic…

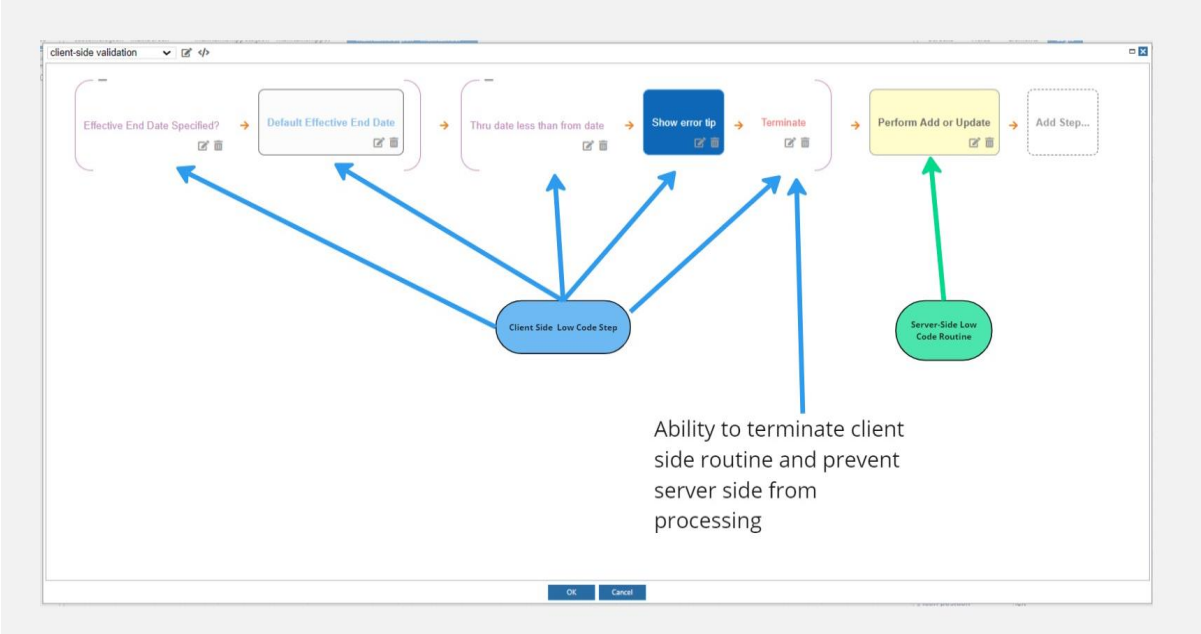


- Give each step a descriptive name. After clicking Add Step…



- Leverage Global/Screen/Session variables
- Leverage our built-in CRUD generator for productivity and a learning tool
  https://profoundlogicsupport.atlassian.net/l/cp/J3pj8yYK
  https://profoundlogicsupport.atlassian.net/l/cp/J5fdfBjD
  https://profoundlogicsupport.atlassian.net/l/cp/9UiyCCuX

- Leverage the ability to mix client-side and server-side low code steps in the same routine: https://profoundlogicsupport.atlassian.net/l/cp/HsLLgnkF



Best Practices for JavaScript Coding (Client and Server Side)

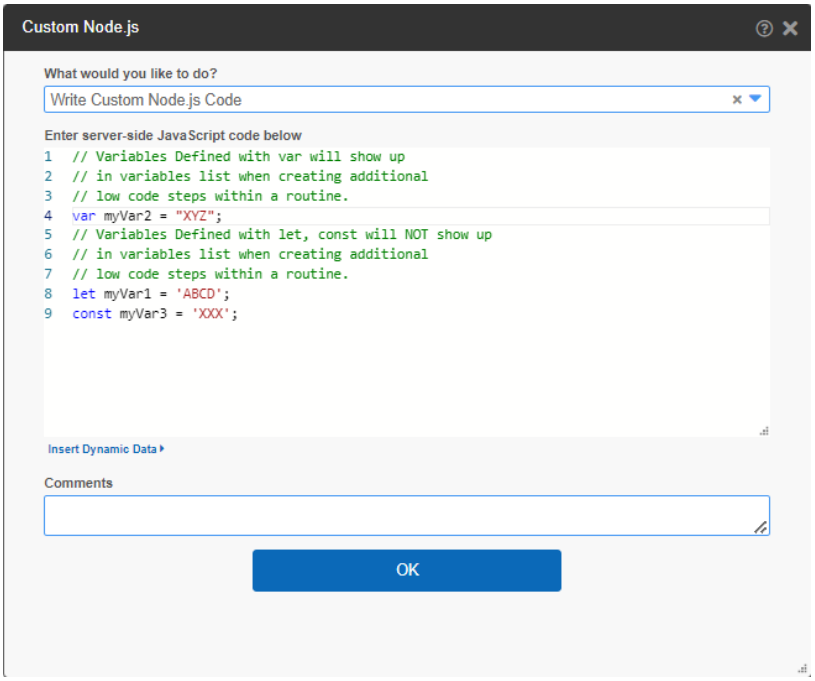
- Use JSDoc to document ALL external JavaScript functions
- Use ESLint JavaScript linting tool used to identify and fix problems in JavaScript code.
  - https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint
- Use namespaces when naming client-side JavaScript functions to avoid potential naming conflicts.

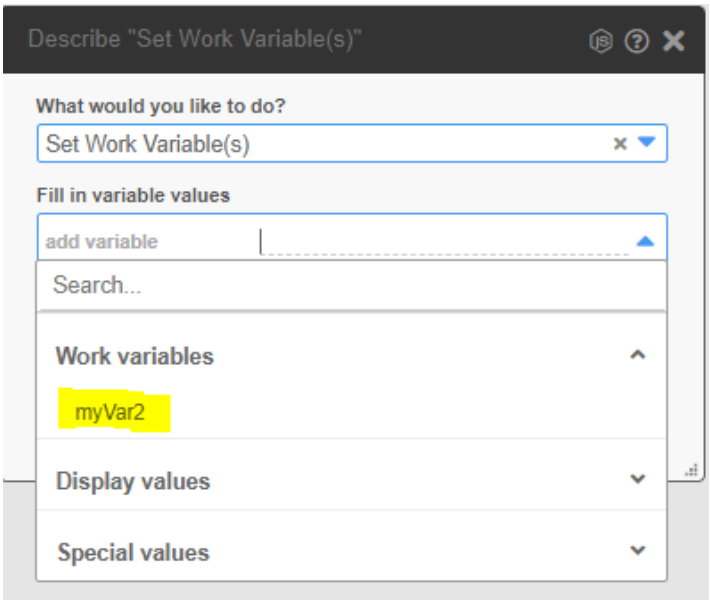
Storing Widget Properties for Database Fields

This feature allows you to associate pre-configured sets of widget properties with database fields and store them for later use. Database fields can then be dragged onto the design canvas, and the associated widget will be created with the pre-configured properties. This allows you to develop database-driven applications more rapidly, since you don't have to continually set up widget properties for commonly used database fields.

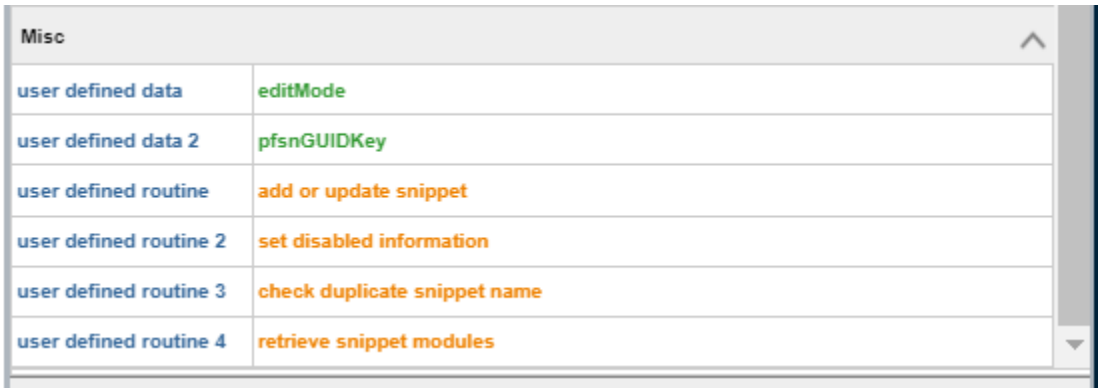
https://profoundlogicsupport.atlassian.net/l/cp/pYnqDm3P

Defining variables in custom low code routines/steps





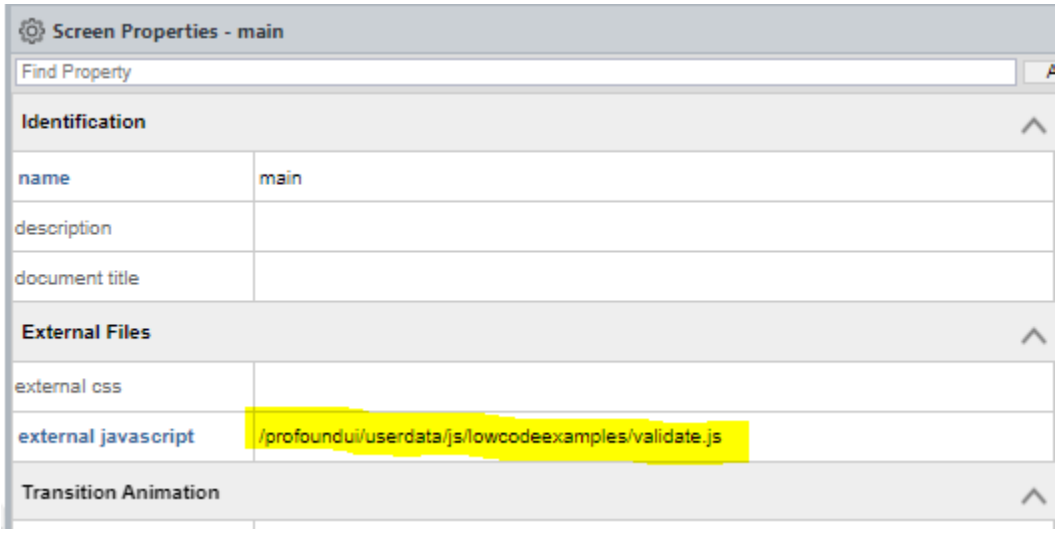
Leverage screen level properties User Defined Data and User Defined Routines

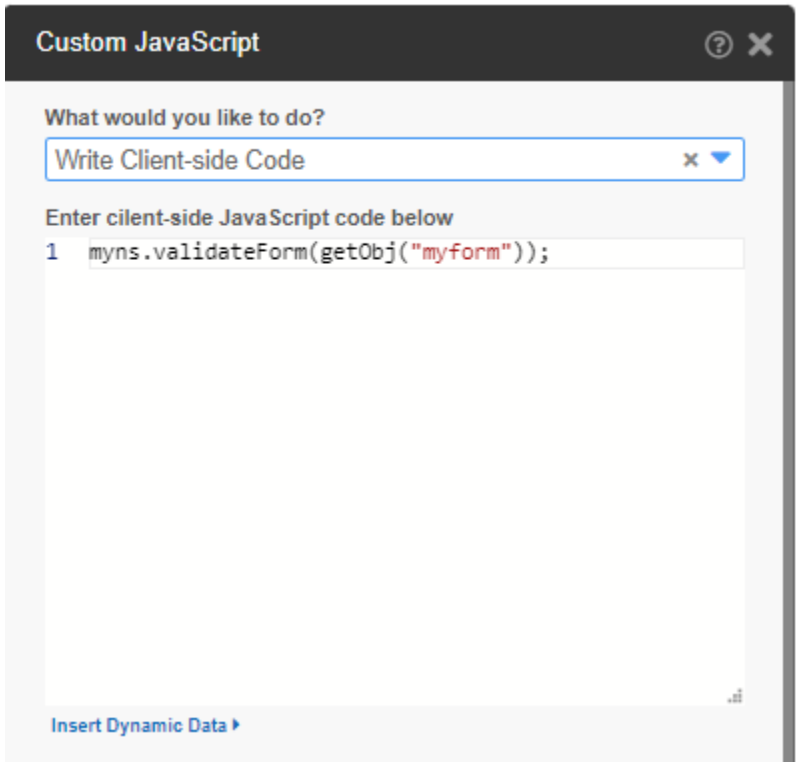


** Attaching low code routines at the screen level will prevent accidental deletion of the routine if all the elements referencing the routine are deleted.

Externalizing custom code

- Define all client-side JavaScript functions in external .js files (userdata) for maintainability. This will provide better maintenance and debugging capabilities.





- Maintain all css classes in external .css files for maintainability

- Use namespaces when naming client-side javascript functions.

```
mynamespace.showGrowlMessage = function showGrowlMessage(severity, summary, detail) {

  // Display Growl Message to User
  $('#growlmsg').puigrowl('show', [{ severity: severity, summary: summary, detail:
detail }]);

};
```

Utilizing Workspaces

A workspace is a convenient way of organizing your projects. A workspace is a directory on the file system that includes all of the source code and configuration files needed to make an application work, along with a Git repository for source control. Workspace directories are stored within the 'modules' sub-directory of the Profound.js installation directory. Workspaces from Profound.js can easily be deployed in the cloud on Profound.js Spaces and vice versa.

When to use a Workspace

Workspaces are best used when your application is a web facing customer interface or a self-contained application that will not communicate with other modules outside of the workspace.

If you are developing integrated applications that must communicate with each other (for example: Warehouse, Inventory, Shipping, A/R, A/P, etc.) then a non-workspace approach will provide more efficiencies regarding integration.

Utilizing External Code

- IBMi Customers – leverage the ability **to call RPGLE programs from a low code module**
  https://profoundlogicsupport.atlassian.net/l/cp/52sKx9m0
- IBMi Customers – leverage the ability **for RPGLE programs to call low code modules**.
  https://profoundlogicsupport.atlassian.net/l/cp/DgrwRWa0
- Make use of low code modules for modularity and reduction of duplicated logic.
  https://profoundlogicsupport.atlassian.net/l/cp/GGLS3mxL

Utilizing other Resources

- Leverage Custom Low Code Plugins.
  https://profoundlogicsupport.atlassian.net/l/cp/URyi2fHF
- Leverage 3rd Party JavaScript Libraries
  https://profoundlogicsupport.atlassian.net/l/cp/8GGcoQ0r

- Utilize Material IO when building application themes https://profoundlogicsupport.atlassian.net/l/cp/qMR7KC5B
- Leverage SQL Layer to reduce complexity.
    - New SQL Tables should use Universally unique identifiers as the primary key and foreign keys. https://profoundlogicsupport.atlassian.net/l/cp/LwU1jKLa
    - Use SQL views to join multiple files to reduce latency.
    - Use CASE/WHEN/THEN to reduce un-needed logic or custom code to format the data.
    - SQL Views: Use 1/0 for indicators to reduce un-needed logic or custom code to format the data.
    - SQL DDL: Leverage Referential Integrity to maintain data integrity.
    - SQL DDL: Leverage Referential Constraints to prohibit data from being corrupted.
    - SQL DDL: Leverage Stored Procedures, UDFs to perform complex batch updates.
    - SQL DDL: Leverage Triggers to perform multi-step processes based on updates to DB.
    - SQL DDL: When creating new databases use SQL Schema (not DDS)

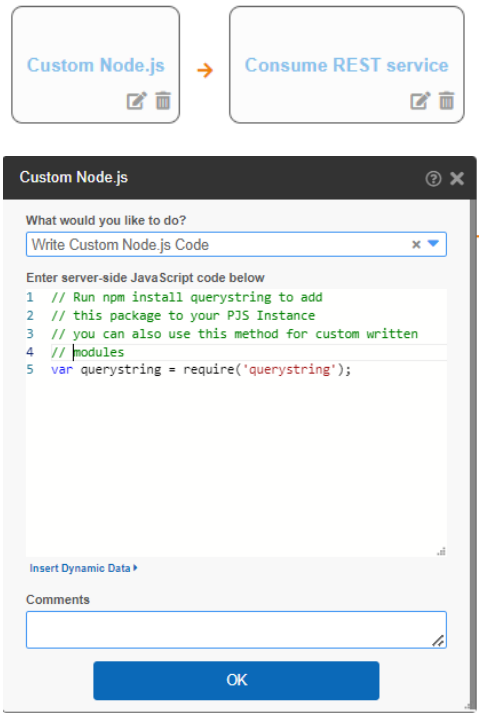
Debugging Low Code

- Use our PL low code debugger (Do not use console.log to debug low code steps) https://profoundlogicsupport.atlassian.net/l/cp/0scQVR0a

Things to Avoid

- Avoid writing lengthy custom code within low code steps - externalize functions and modules. **Server Side:** Leverage external Nodejs packages and custom modules for server-side:

  Using external Nodejs Packages


Custom Node.js
Consume REST service
Custom Node.js
What would you like to do?
Write Custom Node.js Code
Enter server-side JavaScript code below
// Run npm install querystring to add
// this package to your PJS Instance
// you can also use this method for custom written
// modules
var querystring = require('querystring');
Insert Dynamic Data
Comments
OK

Consume REST service

What would you like to do?

Consume REST Web Service

Endpoint URL

```
${encodeURI(process.env.ADOBE_TOKEN_URL)}
```

Insert Dynamic Data ▸ Insert Dynamic URL ▸

Use an OpenAPI Specification - *Optional*

Method

POST

Headers - *Optional*

```
{
    'Content-Type': 'application/x-www-form-urlencoded'
}
```

Insert Dynamic Data ▸

Body - *Optional*

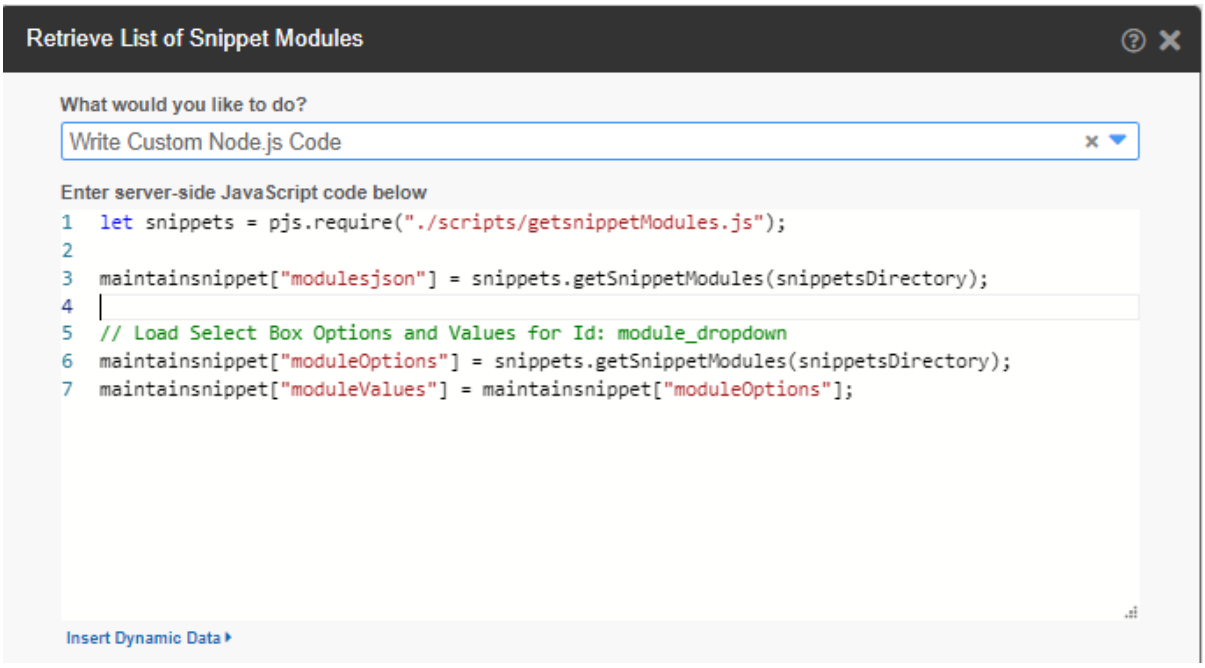
```
querystring.stringify({
  grant_type: 'client_credentials',
  client_id: process.env.ADOBE_CLIENT_ID,
  client_secret: process.env.ADOBE_CLIENT_SECRET
,
  scope: process.env.ADOBE_CLIENT_SCOPE
})
```

Insert Dynamic Data ▸

Use JSON ☑

Auth - *Optional*

Using external custom Nodejs modules



getsnippetModules.js

```
/**
 * Retrieve a list of Snippet Low Code Modules and Routines from a Directory
 * @description Used by maintain.snippets.json to retrieve a list of Low Code Modules and Routines.
 * @module getSnippetModules Retrieve List of snippet module json files from snippet directory
 * @author Jeffrey C. Williams <jwilliams@profoundlogic.com>
 * @param {directory} string - Directory Containing Location of Snippet Low Code Modules
 * @returns {string} - JSON String of Modules and Routines (JSON.stringify())
 * @requires module:fs
 * @example
 * // Retrieve List of Low Code Modules and Routines in a Directory
*  let snippets = pjs.require("./scripts/getsnippetModules.js");
*  let modulesList = snippets.getSnippetModules("./modules/snippets")
*/
const fs = require('fs');

let snippetModules = [];

function getSnippetModules(directory) {

  snippetModules = [];

  // Retrieve List of module files in Directory
  const dirents = fs.readdirSync(directory, { withFileTypes: true });

  const moduleNames = dirents
    .filter(dirent => dirent.isFile())
    .map(dirent => dirent.name);
```

```
    // Retrieve Routines of each module file
    moduleNames.forEach(module => {
      pathName = directory + "/" + module;
      const data = fs.readFileSync(pathName, {encoding:'utf8'});
      processModuleFile(module,data);
  });

  return JSON.stringify(snippetModules);
}
```

Select Box Properties - module_dropdown

Find Property | Set

id	module_dropdown
widget type	select box
value	pfsnmodulename
disabled	isViewMode
required	true
choices	moduleOptions
choice values	moduleValues

- Avoid converting low code steps to JavaScript.
    - This will help minimize any manual code re-factoring that would otherwise be required when upgrades occur.
    - Hint: Create a duplicate of a low code step and convert it to look at the generated code for learning or insights (but delete it after). Once you convert a low code step to javascript, it cannot be converted back to a low code step.

Staying current on product releases

Make every effort to stay current on Profound Logic product releases. New enhancements are continually introduced into our product base. Often, these enhancements are a direct result of customer requests.

Other Design Recommendations

- When submitting jobs to batch from low code, use a monitor subsystem job/message queue/database table. This will remove/reduce latency issues and provide for a more stable and auditable process.

- Follow Loosely Coupled System Design Patterns.
- Use GNU Make to deploy from Git Repo to target instance.