#WIKI - TOC

Section	Descriptions
LEADS Overview	Central location for Lithia Enterprise Application Development & Support (LEADS) initiatives, operational standards, and documentation.
Engineering Operations	Central location for LEADS Engineering Operations standards, best practices, and work flows.
Agile Operations	LEADS Agile practices and workflows are documented here.

##Engineering Overview TOC: | Section | Description | |--|--| | Engineering Standards | Documentation describing how engineering plans, designs, and executes its work | | Source Control Management | Documentation describing our source control system, how we use it, and the workflows used. | |Production Deployments | Documentation describing our methodology for branching and releasing our software | |Proposals | List of proposals and supporting documentation and decisions made |

#LPO-FPO-FPR Get the count of lien records after the set 'importDate' in milliseconds

```
db.lienPayoffs.count({ importDate: { $gte: 1614096000000 } });
```

Remove lien records after the set 'importDate' in milliseconds (be carefull!)

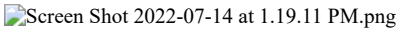
```
db.lienPayoffs.deleteMany({ importDate: { $gte: 1614096000000 } });
```

- Azure Account
- Azure App Service
- Azure Database
- Azure Resources
- Docker
- ES7 React/Redux/GraphQL/React-Native snippets
- Excel Viewer
- GitLens
- Markdown Preview Enhanced
- Prettier - Code Formatter

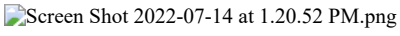
LPO Chart Overview

This LPO chart shows:

- amount of records imported and created for each day
- The ID's and dates that records failed to import

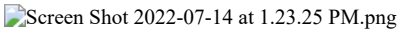


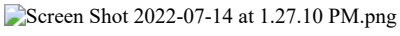
First we pull in the records from a separate "data" folder



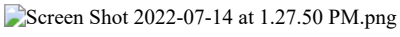
Next we must import in our boilerplate chart.js code

Then we loop through our records to retrieve only the relevant data needed for our chart

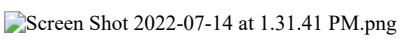




Here we are utilizing those arrays that are holding our data



Lastly, we render out our chart by returning the *Bar* component with our options and data. This is also where we will map through our failed records so that the user knows which days had failed imports.



Topics Covered

Guideline	Description
theme.js	Material UI global theme file, consumed by MUI theme provider
Implementing Styles	Process of adding custom styles to components
classes Vs. className	Differences of and usage of both properties
Target Elements Inside makeStyles()	Use CSS Selectors to specify what elements are styled
Semantic Meaning To Properties	Prop naming guidelines
Component Directory Architecture	Where to put components built for usage throughout application
Avoid Inline Styling	Reasons not use inline styling
No Need For CSS Files	Why we don't need css files
Build Components That Dont Break	Supply default props to values that may break our application
Prop Destructuring	How to do it, and patterns to avoid

theme.js

- A good reference of what default styling is provided by MUI is shown in the documentation.
- theme.js should only be used for global style changes.
  - EX: Primary, secondary, tertiary, and font colors; font family font size, global spacing, etc.
- We want theme to provide us with a solid baseline that our components can sit on, but we don't want to be too specific.
  - EX: If we want our buttons to always maintain a minimum width, we don't want to do that in theme, we should define this styling inside that button component.
- If we needed to override a MUI style setting and for some reason can't be done in the component itself, this belongs in the overrides section of our theme.js object.

```
const theme = createMuiTheme({
  palette: {
    primary: {
      main: purple[500],
    },
    secondary: {
      main: green[500],
    },
  },
  overrides: {
    MuiButton: {
      root: { fontSize: "45px" },
    },
  },
});
```

Implementing Styles

- We will be using the useStyles() hook from MUI for most all of our styling, and can be used with normal JSX elements as well. This is because we are moving away from CSS files, and inline styling. Also, by using this method of styling we should always know where and how to change the style of an element in any component throughout the application.
- We will follow documentation best practices by defining a useStyles variable after imports, and assign that to a class value inside our functional components.
  - EX: https://material-ui.com/styles/api/#examples-3
- If we need hand style a component from scratch, MUI offers a styled method as well
  - EX: https://material-ui.com/styles/api/#styled-component-styles-options-component

---

classes Vs. className

- className
  - className essentially takes what style changes you want and merges these with root.
  - **Used when the component is being implemented.**
- classes
  - Where classes shine is getting hands on with nested components. Looking at the documentation on Button, you can see all the css class names that are available for styling.
  - Using classes, we can do something like classes={ { root: classes.root, startIcon: classes.startIcon } } to provide styling to the starting icon without providing any styling inside that icon component itself.
  - **Used when the component is being created.**
- When structuring our JSX, we don't want to give the developer too much control, but it is a good idea to have a backup method to supply needed custom styling. We do this by specifying our className before our classes when **creating** our component. If we were to specify className after, that would mean our changes when **implementing** our component may be overwritten by some default style settings.

EX: Component built as a tool (Component in Component Library)

```
<Button
  variant={variant}
  className={className}
  classes={{ root: classes.root }}
  onClick={onClick}
/>;
```

EX: Component Implemented in Application

```
<Button
  variant="contained"
  buttonText="Button"
  textColor="red"
  buttonColor="black"
  className={classes.button}
/>;
```

Target Elements Inside makeStyles()

- Sometimes it may be difficult to target the exact element inside a component you wish to style. One way is to use CSS selectors to target which element you wish to apply styles to.
  - EX: Input inside a MUI Text Field

```
const useStyles = makeStyles((theme) => {
        return {
                textField: {
                        '& input': {
                                textAlign: 'left',
                        },
                },
        };
});
```

```
<CurrencyTextField
  {...input} // Name, onBlur, onChange, checked, ...etc
  {...rest} // Similar to props, but some differences > name, label, outlined, variant, id, ...etc
  label="Amount"
  variant="outlined"
  value={input.value}
```

```
      currencySymbol="$"
      outputFormat="string"
      modifyValueOnWheel={false} // Will break app when scrolling while input is focused (see autonumeric docs)
      onChange={(event, value) => {
        input.onChange(value);
      }}
      error={meta.touched && !!meta.error}
      helperText={meta.touched && meta.error}
      InputProps={{
        className: classes.textField,
      }}
    />
```

Semantic Meaning To Properties

- We should avoid plugging direct terms to prop names.
  - EX: color='red'
  - In CSS, color means font color, where backgroundColor means… background color. Changing these property names to be textColor='red' and buttonColor='black' provides a clear meaning to each property defined.
- It's not a hard and fast rule, because people may put meanings to properties that others might not agree on, but we want to make these properties clearly defined, easy to plug and play, with out much back and forth through documentation.

Component Directory Architecture

- Currently without a dedicated library, we need to separate our tools from our actual creations. We are using a directory inside of src/components called _component_library where we keep these tools.
  - EX: We want our Approval button to have different colors based on its status.
  - We can leverage our Button made in the library, and use javascript to determine what color we want that button to be without changing any of the source code in the button
- With bigger more unique components like our KendoGrid, because it is so specific and would take a ton of time to parse out all the individual pieces to be to the _component_library, we can leave all of that in its own file. If some things can be componentized, like a data cell to be reused within our specific grid component, we can extract that to its own file but still within /kendo-grid.

Avoid Inline Styling

- The main issue with inline styling is its precedence. If we change the maximum size of an element in our components class attribute but have that max size specified using inline styling, the inline styling will take precedence over our global style change.
- Another reason to avoid inline styling is we want all our styles defined in one easy to access spot. If we used inline styling here and there, it could become confusing as to where style changes are happening.
- Keeping our styles in makeStyles() also means we always know where to go when style changes are needed.

No Need For CSS Files

- Because we are leveraging MUI makeStyles() or using MUI's version of styled-components, we should not be using any CSS files to define our styling.
- One exception to no CSS files are for global styling not easily achieved using our theme.js file. Examples here could be font family, removing default element margins, etc.
- To reiterate, we should not be using CSS files, and if we must use so only when absolutely necessary.

Build Components That Dont Break

- The end goal is to build a component you can drop in any react application, and it should just work. That means supplying default props if needed, some template text as a starting point, things like that.
  - EX: const { onChange = () => {}, buttonText = 'Example Button Text' } = {…props}
  - We need an onChange event for our input, we can provide the above line at the top of our component.
  - That way if the user starts typing, it doesn't try to call a method that is undefined, is not a function, etc. and breaks our application.

Prop Destructuring

- What this means is if we are going to be using only parts of our props object, just split them out and define them at the top of the component. That makes it clear exactly what pieces we are using from these props.
  - EX: How To Destructure
  - const { locked, value, isLoading } = props;
  - EX: How Not To Destructure
  - props.locked, props.value, props.isLoading
- Break props down inside the component, instead of destructuring them in the arrow function itself. This sets up a paradigm of how we will be handling objects in JavaScript, and it confusing if we pass in an entire {...rest} object that we prop drill into our component.
  - EX: How To Handle Props
  - const ExampleComponent = (props) => {...}
  - EX: How Not To Handle Props
  - const ExampleComponent = ({isLoading, value, ...rest})) => {...}
- Important: If we do need every part of a property passed into another component, make inline comments describing why we aren't breaking the object up and what its purpose is.

EX: Passing

```
const GeneralModal = (props) => {
  const {
    onDismiss = () => {},
    mainCallback = () => {},
    content = 'Add some text content, supports JSX',
    title = 'General Modal Title',
    showModal,
    submitText = 'Ok',
    cancelText = 'Cancel',
    noSubmit = false,
    noCancel = false,
    noButtons = false,
```

```
  } = { ...props };
  return (
    <Modal
      onDismiss={onDismiss}
      content={content}
      title={title}
      open={showModal}
      actions={
        !noButtons && (
          <>
            {!noSubmit && (
              <Button
                onClick={mainCallback}
                variant='contained'
                color='primary'
              >
                {submitText}
              </Button>
            )}
            {!noCancel && (
              <Button autoFocus onClick={onDismiss} color='primary'>
                {cancelText}
              </Button>
            )}
          </>
        )
      }
    />
  );
```

EX: Not Passing (Without comments, what values are ...rest, ...input passing?)

```
const SelectInput = ({ label, labelWidth, className, options, ...rest }) => {
  return (
    <Field
      render={({ input, meta, ...rest }) => (
        <FormControl
          size="small"
          variant="outlined"
          className={className}
          label={label}
          error={meta.touched && !!meta.error}
        >
          <InputLabel id="form-group-name-input-label">{label}</InputLabel>
          <Select
            input={<OutlinedInput labelWidth={labelWidth || 50} />}
            {...rest}
            {...input}
          >
            {options}
          </Select>
          {meta.touched && !!meta.error && (
            <FormHelperText>{meta.error}</FormHelperText>
          )}
        </FormControl>
      )}
      {...rest}
    />
  );
};
```

---

https://www.youtube.com/watch?v=ML5egqL3YFE

Section	Description
Git Workflow	
Hotfix Scenarios	
Hotfix Workflow	
Cutting a Release	
Notes about Merging Branches	
Additional Information & Resources	

Git Workflow

graph TD; A[Develop] --> B[release/v1.2.3] D --> B B --> D[hotfix/abarnes-23456] A --> I[feature/abarnes-45678] I --> A A --> J[bug/abarnes-45678] J --> A B --> F[TEST] B --> G[UAT] B --> H[MAIN]

#Hotfix Scenarios

//content goes here

#Hotfix Workflow

//content goes here

#Deploy Develop into Test

Creating a release branch

Release branches are created from the develop branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of develop is ready for the “next release” and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

```
$ git checkout -b release/v1.2 develop
Switched to a new branch "release-1.2"
-- bump version in future (Currently working on that...)
$ git commit -a -m "Bumped version number to 1.2"
[release/v1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

Edit Readme.md

Change the version number to the release version (1.2) and save the file.

Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out.

The newly created release branch should be merged into the test branch

```
$ git checkout test
Switched to branch 'test'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2 -m "test version 1.2"
```

Deploy to the test environment

- Work with the QA team members to schedule a deployment time where it wont disrupt their current testing.
- Once deployed to the test environment, please publish the release notes and notify the QA team.

The release is now done, and tagged for future reference.

#Deploying to the UAT Environment

Merge test into UAT

Merge the test branch into UAT and tag the associated version. If you don't know the current version use `git tags`. Please also tag the test version with the current test version.

```
$ git checkout uat
Switched to branch 'uat'
$ git merge --no-ff test
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2 -m "uat version 1.2"
```

Deploy to the uat environment

- Work with the QA team members and functional to schedule a deployment time where it wont disrupt their current testing.
- Once deployed to the uat environment, please publish the release notes and notify the QA team.

Notes about merging branches

- Be sure to pull the latest changes and merge them into your feature branch before pushing the branch and merging it into the parent branch.
  - This will avoid merge conflicts in the parent branch
- Please be sure that all your tests are passing before merging your code into the parent branch
  - This is to avoid broken builds
  - In the future, we will introduce test automation that will reject merging feature branches if tests do not pass.

Additional Information & Resources

For more information on tagging please see https://git-scm.com/book/en/v2/Git-Basics-Tagging For more information about semantic versioning please see https://semver.org/

#Git Branch Management

`Type/name-number`

Developer-Name

- first initial of first name and last name
- example: Andy Barnes = 'abarnes'

Type

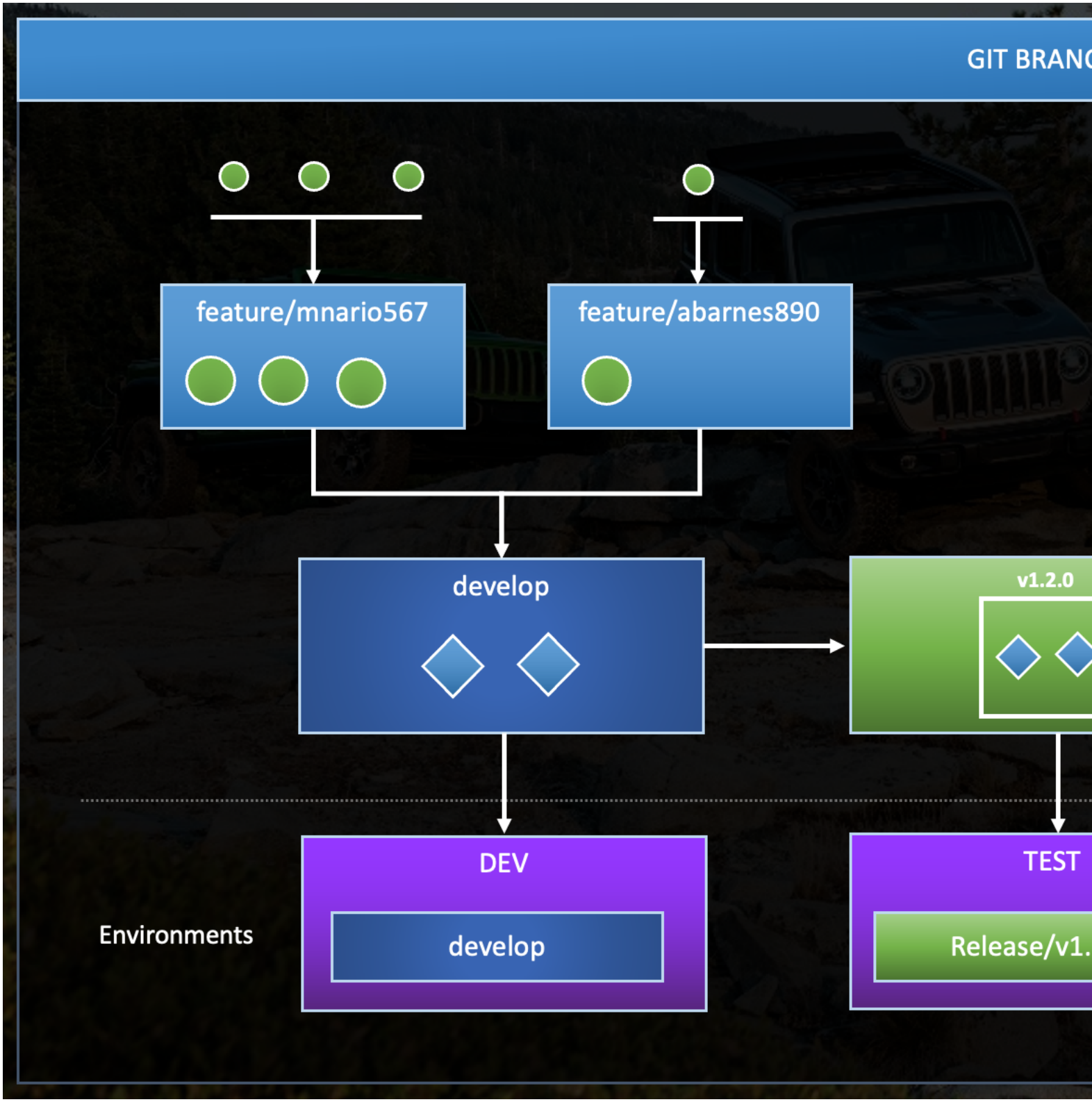
- The type of work outlined in DevOps
  - feature (user story)
  - bug
  - hotfix
  - release

Item Number

- The item number associated in the DevOps sprint
- If there is not an item number associated, use a description key word ('demo', 'test')

Branch naming examples:

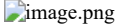
- feature/abarnes-1234
- bug/v1.2/wsmith-3456
- bug/abarnes-12345
- hotfix/v1.0/jadams-1232
- release/v1.2

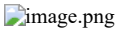


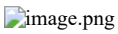
1. Select `Import Repository` from the repo dropdown list

image.png

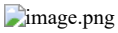
2. Supply original repo clone URL, and generated username/password







3.) Go to the new repository, click branches, make `main` the default branch. Removing old branches is OK to do as well.



4.) OPTIONAL - If you want to fork with just the `main` branch, create a fork off the desired repo and select the option to just have the main branch available. Then follow the `Import Repository` steps listed here for that forked branch.

Summary Overview

1. All production shall be submitted in the CAB either as a standard change or emergency change. Please see CAB Review Process
2. All production deployments shall follow the Deployment Checklist

Note: This checklist is Application specific. There will be a future checklist for ETL deployments | Order |Description | | |--|--|--| | 1| Inform the ETL team with our intention to deploy version --- Be sure to get their signoff and that someone is on standby to validate the release changes| TPM | | 2| Inform the DevOps team with our intention to deploy version --- Be sure to get their signoff and that someone is on standby just in case we need support| TPM | | 3| Inform the Business team with our intention to deploy version --- Be sure to get their signoff and that someone is on standby just in case we need support| TPM/BA | | 4 | Schedule a date/time of the release and be sure to inform the key members of this event | TechLead | | 5 | Create a change request for CAB - If the deployment is before the CAB weekly review please submit this as an emergency change | TPM/TechLead | |6 | Create a new thread in the General channel "Production release v1.0.0" and announce our intention to release updates into production | TPM/TechLead | |7 | Update the Environment Release Version to reflect the version changes in the production environment | TechLead | |8 | Merge uat branch into main branch to trigger CI/CD pipeline deployments to PROD. Monitor app and database pipelines through to completion. | TechLead | |9 | Update the environment summary, deployment history, and release notes of what was deployed in Wiki | TechLead | |10 | Inform the General Teams channel that production was update to the new version and provide the release notes | TechLead/TPM | |11 | Get conformation from the post production validation team members that the changes are working as expected | TechLead/TPM |

##Deployment Procedures

Order	Description	
1	Team members mark work as QA ready on the DevOps board	DevOps
2	Arrange QA testing - if testing is successful it is ready to be promoted to production. If testing isn't successful a bug should be created to address a fix and then re-tested to promote to production.	BA
3	Assign tag to work items that are ready for UAT prior to being promoted to production.	TechLead/BA/TPM
4	Arrange UAT testing of all items being promoted into production and track and document issues.	BA
5	Create wiki page with release information to include items included in the release and the status of QA and UAT.	TPM
6	BA communicates results of UAT testing and TechLead promotes all approved items into production.	BA/TechLead
7	Communication is sent out to Stakeholders that the release is now live in production with the release notes.	TPM

CAB Review process

Starting March 31st 2021 Production deployments shall be submitted via Change Approval Board (CAB).

- This does not mean that you can't deploy changes to production. I means that we have to file the change with CAB
- Normally, if we have a scheduled deployment we shall submit a change request ahead of time so that we can review the change before it happens.
  - This is normally submitted as a standard change.
- Abnormally, if we have a change that request deployment before the next CAB review, we retroactively file that request.
  - This is submitted as an emergency change
- Create a new Change request here

CAB Review Template

- Type = Emergency
- Assignment Group = Application Development
- Assigned to = Andy Barnes
- Short Desc - Production release version
- Long Desc - Release notes available here
- Planning - Submit the release process
- Justification - Submit the reason for the change
- Implementation Plan - Prod release checklist
- Backout Plan - Rollback version to 1.2.19
- Test Plan - Functional team and BA will review post deployment steps and validate the changes and environment stability
- State = New
- Risk = Low
- Impact - Single to Limited Users
- Priority - Pre-Approved
- Schedule
  - Start – Thur 02/11/21 at 4:30 pm PST
  - End – Thur 02/11/21 at 5 pm PT
  - Cab require - No
  - CAB Date 02/12/21
  - Cab Delegate - Andy Barnes