Progress

- **Lid_on_top error handling**

To counter the problem of carefully managing the placement of the lids over the labwares, we devised a way so that the Jubilee API can store the location of all the lids. In this way, even if we run some erroneous commands like gripping a lid from a labware which has no lid on top, then the API will pop out an error message.

To implement the above functionality, we added an attribute called lid_on_top in the Labware class. This attribute gets updated whenever we pick and place lids using the vacuum gripper tool.

The Jubilee API can store the locations of the lids, we added an error handling function which can be called whenever we need to error check if the lid is there or not.

```
def lid_on_top_error_handling(self, location: Union[Well, Tuple, Location, Labware], expected_condition: bool):
    """Raise an error if the lid is on top of the tool."""

    # Error handling to check if the labware at location has lid or not
    if isinstance(location, Well):
        labware = location.labware_obj
    elif isinstance(location, Location):
        well_obj = location._labware
        labware = well_obj.labware_obj
    elif isinstance(location, Labware):
        labware = location

    bool_override_choice = True

    if labware.has_lid_on_top == expected_condition:
        pass
    else:
        if labware.has_lid_on_top == True:
            error = self.OverridableError(f"Lid is on top of {labware}")
            # raise ToolStateError(f"{labware} Labware has lid on top")
            bool_override_choice = error.ask_override()
        else:
            error = self.OverridableError(f"Lid is not on top of {labware}")
            # raise ToolStateError(f"{labware} Labware has no lid on top")
            bool_override_choice = error.ask_override()


    if bool_override_choice:
        pass
    else:
        raise error
```

- **Liquid_level_check**

To counter the problem of overflow/underflow of liquid out of the vials/beakers, we needed some way to store the current liquid volume in the vials.

To implement this, we added an attribute called currentLiquidVolume in the Well class which can be used to store the current liquid volume in every well object like the sample_vials or the solvent beaker.

This currentLiquidVolume attribute of the Well class object gets updated on every aspirate or dispense operation with the dual syringe or the single syringe tool.

```python
def update_currentLiquidVolume(self, volume: float, location: Union[Well, Tuple, Location], is_dispense: bool):
    """Update the current liquid volume for the well at location."""

    # Get the well object from the location
    if isinstance(location, Well):
        well_obj = location
    elif isinstance(location, Location):
        well_obj = location._labware

    # Update the current liquid volume for the well
    if is_dispense == True:                    # volume is dispensed at the location
        well_obj.currentLiquidVolume += volume
    else:
        well_obj.currentLiquidVolume -= volume
```

To error check if the vial has enough volume to accommodate more liquid into it or not, we implemented this error check function in the Tool class.

```python
def check_liquid_level(self, location: Union[Well, Tuple, Location], target_volume: float, is_dispense :bool):
    """ Error handling function to check if the well at location has enough liquid to aspirate target_volume
        Or if can accomdate target_volume for dispensing at the well location
    """

    # Get the well object from the location
    if isinstance(location, Well):
        well_obj = location
    elif isinstance(location, Location):
        well_obj = location._labware

    bool_override_choice = True

    # Check if the well has enough liquid to aspirate or dispense the target_volume
    if is_dispense == True:                    # volume is to be dispensed at the location
        if well_obj.currentLiquidVolume + target_volume <= well_obj.totalLiquidVolume:
            pass
        else:
            error = self.OverridableError(f"{well_obj} Well cannot accomodate {target_volume} ml dispense liquid volume ")
            # raise ToolStateError(f"{well_obj} Well cannot accomodate {target_volume} ml dispense liquid volume ")
            bool_override_choice = error.ask_override()
    else:                                      # volume is to be aspirated out of the location
        if well_obj.currentLiquidVolume - target_volume >= 0:
            pass
        else:
            error = self.OverridableError(f"{well_obj} Well does not have enough liquid to aspirate {target_volume} ml liquid volu
            # raise ToolStateError(f"{well_obj} Well does not have enough liquid to aspirate {target_volume} ml liquid volume ")
            bool_override_choice = error.ask_override()

    if bool_override_choice:
        pass
    else:
        raise error
```

- **Uploaded Axo's pics/ videos and CADs on One Drive folder**

- **Uploaded latest code on Axolotl git repository**

- **Spectrum data management**

We added code to the OceanDirectAxo.py i.e the spectrometer tool API. This code is used to store all the spectrum data in a more organised manner.

This is the directory structure in which the spectrum gets stored. Suppose the experiment gets conducted on date : 25-07-25 by two individual operators named: Aditya, Hardik. Now Aditya did two different experiments named : expt1, expt2 , while Hardik did an experiment named : expt3.

```
\Spectrum_data
    \25-07-2025
        \Aditya
            \expt1
                experiement_metadata.json
                mof_recipes.json
                \references
                    dark.npy
                    white.npy
                \spectra
                    A1.csv
                    A1_T0_absorbance_spectrum.png
                    A1_T10_absorbance_spectrum.png
                    A1_T20_absorbance_spectrum.png
                    A2.csv
                    A2_T0_absorbance_spectrum.png
                    A2_T15_absorbance_spectrum.png
            \expt2
                experiement_metadata.json
                mof_recipes.json
                \references
                    dark.npy
                    white.npy
                \spectra
                    A1.csv
                    A1_T0_absorbance_spectrum.png
                    A1_T10_absorbance_spectrum.png
        \hardik
            \expt3
                \\..\\
```

The contents of the experiment_metadata and mof_recipes.json can be accessed here.

To implement the above data management plan, we added extra code during the Spectrometer tool initialization. These extra initialization arguments we used to make the directories in the name of the operator and the experiment being performed.

```
# Load the spectrometer

spectrometer = Spectrometer(index = 3,
                            name = 'Spectrometer',
                            base_dir=r"C:\Axo\science-jubilee\axo\spectrum_data",
                            experiment_name= 'ZIF-67 Synthesis',
                            operator_name= 'Aditya_and_Hardik',
                            target_compound='ZIF-67',
                            # project_id = 'MOF_Phase1',
                            experiment_notes= 'Synthesis of a single vial of MOF - First run',
                            solvent = 'methanol',
                            temperature_c= 25)
                            # ref_dark = 'dark_20250708_165121.npy',
                            # ref_white = 'white_20250708_165110.npy')
axo.load_tool(spectrometer)
```

We also added a function in the OceanDirect_axo API called record_mof_recipe. This function is called right before the spectrometer collects the absorbance data from the vial. This function is basically used to dump all the vial data into the mof_recipes.json file.

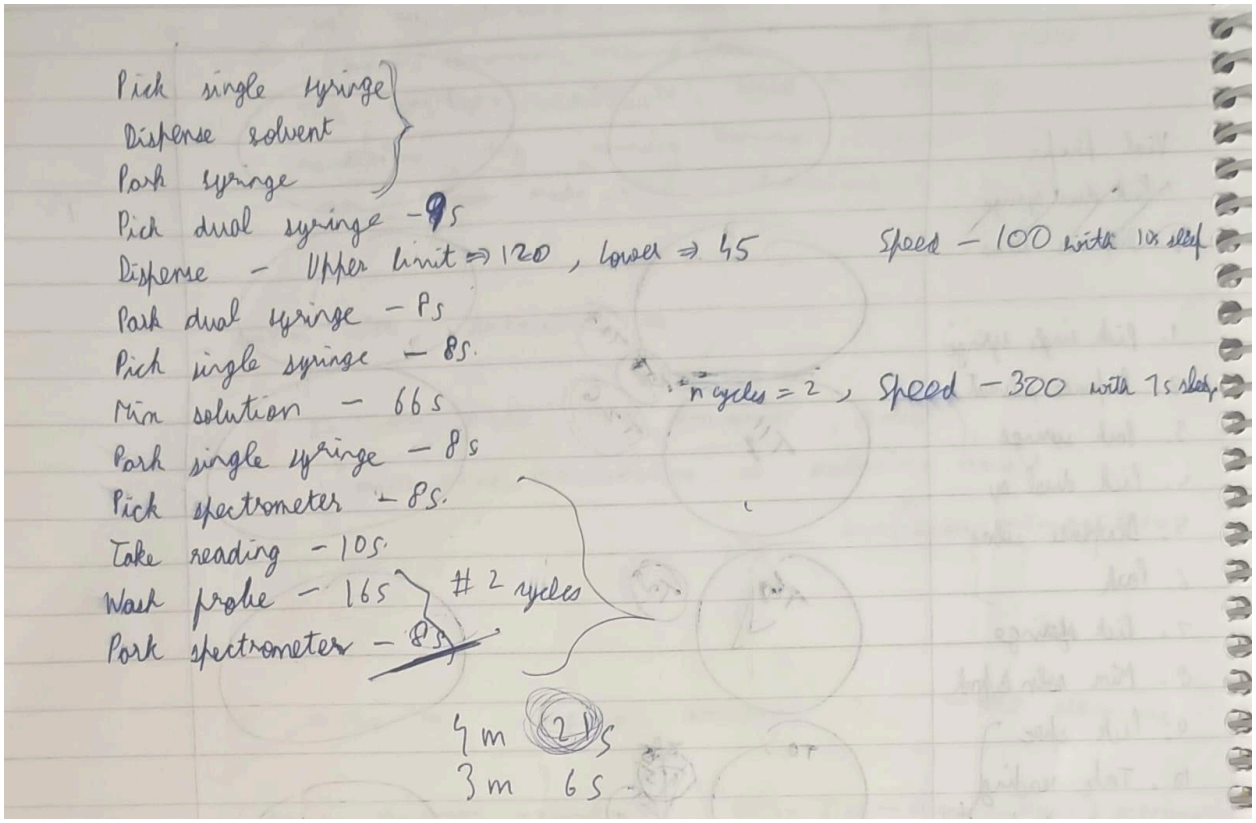
```
class Spectrometer(Tool, OceanDirectAPI):
    def record_mof_recipe(self,
                          well_id: str,
                          metal_precursor_name: str,
                          metal_precursor_vol_ml: float,
                          organic_precursor_name: str,
                          organic_precursor_vol_ml: float,
                          solvent_name: str = "methanol",
                          solvent_vol_ml: float = None,
                          additional_notes: str = None):
        """Store MOF synthesis recipe for a specific well."""

        recipe = {
            "metal_precursor": {
                "name": metal_precursor_name,
                "volume_ml": metal_precursor_vol_ml
            },
            "organic_precursor": {
                "name": organic_precursor_name,
                "volume_ml": organic_precursor_vol_ml
            },
            "solvent": {
                "name": solvent_name,
                "volume_ml": solvent_vol_ml
            },
            "total_volume_ml": metal_precursor_vol_ml + organic_precursor_vol_ml + (solvent_vol_ml or 0),
            # "molar_ratio": f"{metal_precursor_vol_ml}:{organic_precursor_vol_ml}",
            # "preparation_time": dt.datetime.now().isoformat(),
            "notes": additional_notes
        }

        # Store recipe for this well
        self.well_recipes[well_id] = recipe

        # Save all recipes to JSON file
        recipe_file = self.experiment_dir / "mof_recipes.json"
        with recipe_file.open("w") as f:
            json.dump(self.well_recipes, f, indent=2)
```

- **Recorded the physical time taken by every tool's functionality**



These timings were used to better the orchestration workflow.

- **Perfected each tool's Picking up and Parking operations**

We perfected the positions of the tool holders of the tool and also reworked on the heat inserts on the Vacuum gripper & Single Syringe tool holder parts.

While working on the Dual Syringe tool holder assembly, we came to the conclusion that there was an internal friction problem that prevented the tool holder rod from sliding-in into the tool. Hence we applied some WD-40 lubricant on the rod and finally it worked.

Tested the picking up and parking of all the four tools 20 times with 100% success rate.

- **Orchestration workflow**

Wrote the orchestration workflow in the Experiment.py file.

Basically before starting the experiment, we provide all the labwares, deck and machine objects to the Experiment class object.

```
# making a list of all the above loaded labware, tools and Machine objects
all_labwares = {"slot1": {"precursors": precursors},
                "slot2": {"samples2_ssy":samples2_ssy, "samples2_sy": samples2_sy, "samples2_spec": samples2_spec},
                "slot3": { "solvents": solvents},
                "slot4": {"vacuum_location": vacuum_location},
                "slot5": { "samples5_ssy": samples5_ssy, "samples5_sy": samples5_sy, "samples5_spec": samples5_spec} }

all_tools = {"single_syringe": single_syringe, "dual_syringe": dual_syringe, "spectrometer": spectrometer, "gripper": gripper}

from science_jubilee.Experiment import Experiment

# Initialising the Experiment
exp = Experiment(machine= axo, deck = deck, all_tools= all_tools, all_labwares= all_labwares)

# Running the experiment
exp.make_batch(r"C:\Axo\science-jubilee\axo\axo_api_testing\draft_synthesis_plan.json")
```

Then after initialization, we call the make_batch function of the Experiment class to start the physical experiment.

While writing the code for the orchestration workflow we followed the following experiment procedure.

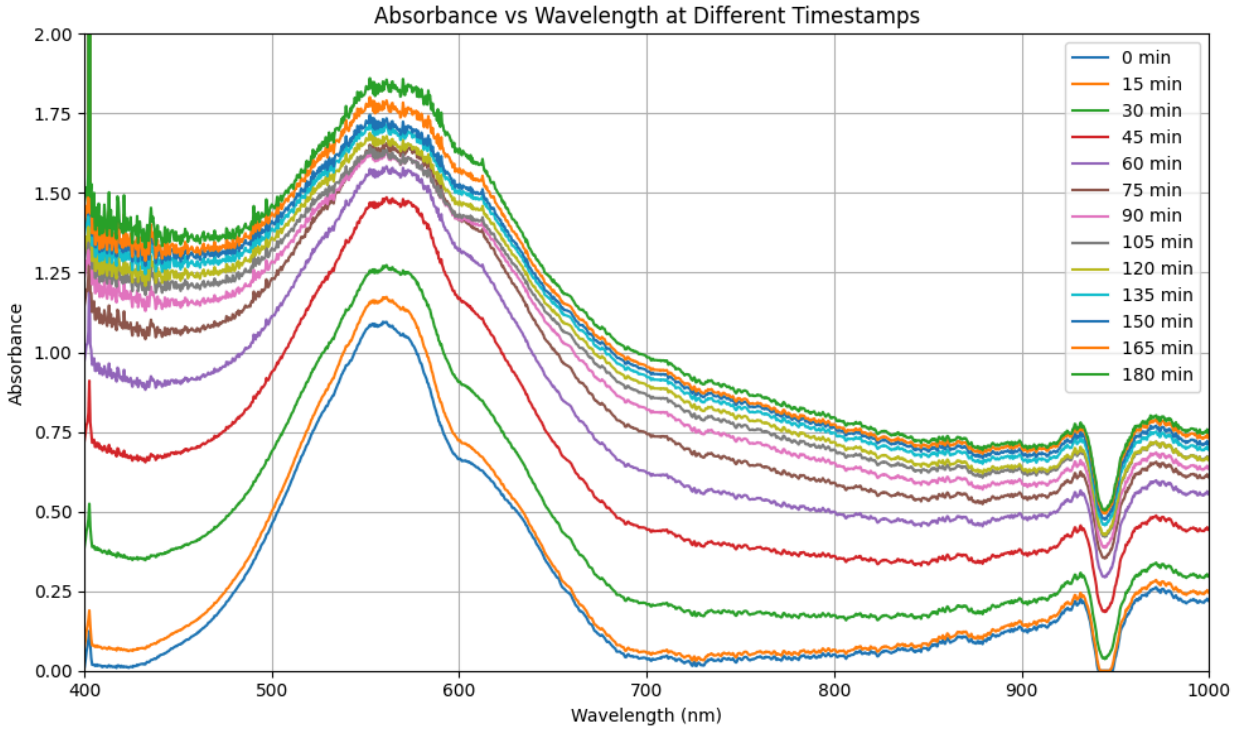
```
1. Fill dual syringe with precursors
2. Dispense solvents in all the vials one-by-one
3. Dispense metal_precursors in all the vials one-by-one
4. Apply make_vial on all the vials one-by-one
        4.1 dispense Organic precursor
        4.2 mix the soln
        4.3 Take T0 spectrum reading
        4.4 wash the probe
        4.5 (Optional) if some earlier vial has
next_spectrum_reading to be recorded, it will do that first and then
continue on this loop
 5. Record spectrum readings at regular intervals for all the vials
```

The full code for the orchestration workflow can be found here.

- **Configured Jubilee on Lab's laptop and Ran Single Vial MOF workflow.**

In the experiment we synthesized a single MOF sample of 10ml total liquid volume. The experiment involved taking 13 readings at an interval of 15 mins each. Overall, the experiment ran for 4 hours with no human interference at all.

The experiment workflow and the absorbance readings can be seen here.



Work to be done

- Make internships reports (Indivisually)
- Share the intern progress form with Prof.
- Write up a maintenance manual document.
- Synthesise a batch of 5 identical MOF samples and show the repeatability of their K_n's values.
- Print the new lids and just replace them. Or else, pass-on the CAD files to Shuang.
- Work with Shuang for the Off-boarding process. [Done]
- Do for the Axo Code documentation.
- Write a Well volume visualisation function in the Experiment class. This function can be called to return an image which showcases all the vials and their currentLiquidVolume written beside them.
- Discuss with Shuang how the Bayesian optimiser inputs can be passed on to the Orchestration code. And accordingly make changes to the Experiment class.

More work

- Write the data log time besides the Reaction time in the spectrum.csv
- [Done] Record the log time of every operation in the orchestration workflow cell and also output all the log times in a separate file.
- Use the spectrum data to check if the k_n match for the samples and share it on slack.

- Design a AI agent which can do the following
    1. Early stopping decision for the vials(can be hardcoded!)
    2. Motion planning agent: The agent should devise the most optimised/efficient way to do the individual ops
    3. Decide on the decision whether to make another vial or record spectrum