Automating data transfer to the ATLAS ITk Production Database

Leandro Rizk
leo.rizk@mail.utoronto.ca
University of Toronto

Supervisors: William Trischuk, Richard Teuscher

19 August 2022

Abstract

The high collision rates of the High-Luminosity Large Hadron Collider (HL-LHC) will require an upgrade for its experiments to perform adequately. The ATLAS experiment's current detector will be replaced with a new Inner Tracker (ITk), with installation starting in 2026. Components that will come together to become the ITk are currently being built around the world and their location and status are tracked on the ITk Production Database. The University of Toronto and its partner Celestica use a Google Spreadsheet to document their progress and their tests relating to the ITk components they are producing. A channel is necessary to efficiently transcribe the relevant data from that spreadsheet to the ITk Production Database. Along with my collaborator Ruchi Soni, I have developed six Python scripts that upload necessary information to the Database. Three of these scripts are for relaying information about the assembly status of components, while the other three are for uploading different test results. The programs written are successful and promise to be useful. The biggest strength of the programs is that, while they perform one upload at a time for each component, they will not crash when encountering a problem with an upload and continue to perform subsequent uploads, all while tracking successful transmissions. A major weakness of all the scripts is their heavy reliance on the structure and contents of Celestica's spreadsheet—changes or errors in the spreadsheet can impede their performance. This report focuses more on the test upload scripts (hybrid glue weight test, module glue weight test, and hybrid wire bonding test) while Ruchi Soni's report focuses on assembly scripts.

1 Introduction

1.1 High-Luminosity LHC

The Large Hadron Collider (LHC)'s crowning achievement was on July 14th, 2012, when its two major experiments, ATLAS and CMS, both reported on measurements of a particle consistent with the Higgs boson [1, 2]. At the time of this discovery, the LHC was at its first run, with proton-proton collisions at 7 TeV of centre-of-mass energy, and was reaching an integrated luminosity of about 30 fb^{-1}. Ten years later, the LHC has now started its third run, with centre-of-mass energy at 13.6 TeV and an integrated luminosity expected to reach 450 fb^{-1}. Successive upgrades to the LHC are required to observe more elusive events at increasingly higher energies and to gather much more data thanks to greater collision rates. The current timeline foresees a major upgrade for the LHC, dubbed the High-Luminosity LHC (HL-LHC). The particle accelerator is planned to collide protons at 14 TeV and, with its intensified collision rates, it will reach an integrated luminosity of up to 4000 fb^{-1} by the end of its next run [3]. The fourth run is planned to begin in 2029, after three years of shutdown ("Long Shutdown 3"), during which the upgrades of the LHC and its experiments are installed.

In Run 4, the Inner Detector currently inside the ATLAS experiment will not be able to sustainably perform under the very high collision rates of the HL-LHC [3]. An upgraded detector will need to be more resistant to radiation and have faster readout channels to handle the higher trigger rates.

ATLAS's Inner Detector will be completely replaced with a new Inner Tracker (ITk) [3], shown in Figure 1. Construction of the parts that will become the ITk is ongoing and is being handled by different institutions around the world.

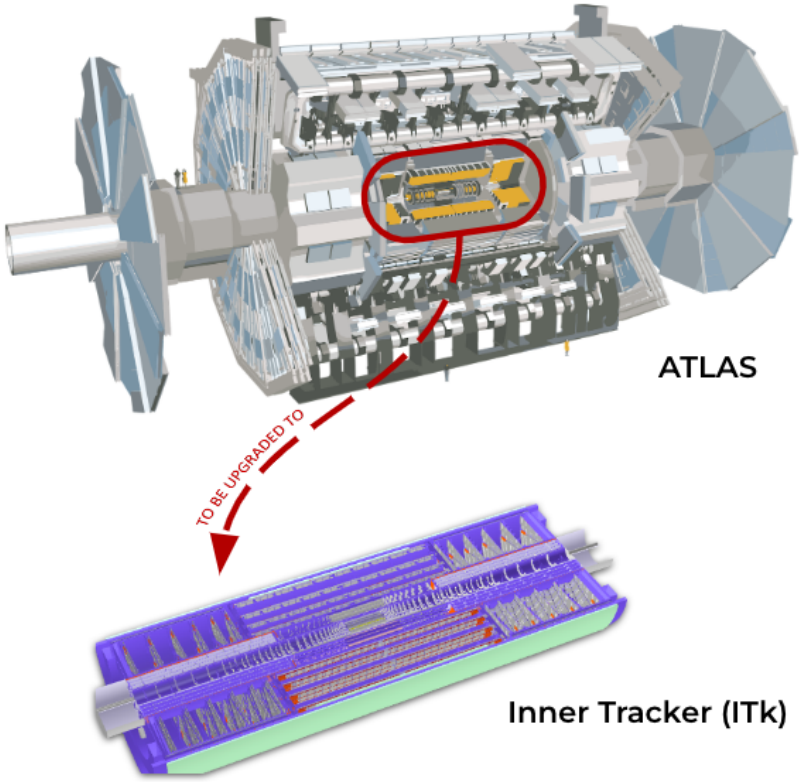

Figure 1: Conceptual design of the Inner Tracker (ITk) and its place inside the ATLAS experiment.

1.2 ATLAS ITk upgrade

Just like its predecessor, the ITk will be the innermost layer of the ATLAS detector and its purpose will be to track the paths of charged particles produced by collisions happening at the centre of ATLAS [3]. The ITk will bear all-silicon high-resolution sensors and will be composed of two main constituents, the "Pixel Detector" (closest to the centre) and "Strip Detector" (surrounding the Pixel Detector) [3]. Figure 2 shows the composition of the ITk.

The Strip Detector is further divided into a barrel and two endcaps [3]. The barrel is composed of four coaxial cylinders, each made of a number of "staves". Staves are double-sided 1.4-metre-long rectangular slabs that hold silicon sensors and electronic circuitry. The main subcomponent of a stave is called a module. Each side of a stave is composed of 14 modules. The modules of two inner cylinders are categorized as "short strip" (SS) while those of the outer two are "long strip" (LS). Each endcap is composed of six disks sitting at one and the other end of the cylindrical barrel. The disks are divided into double-sided, identical petals, about 0.6 metres long. All petals have the same 6 modules, labelled R0 to R5.

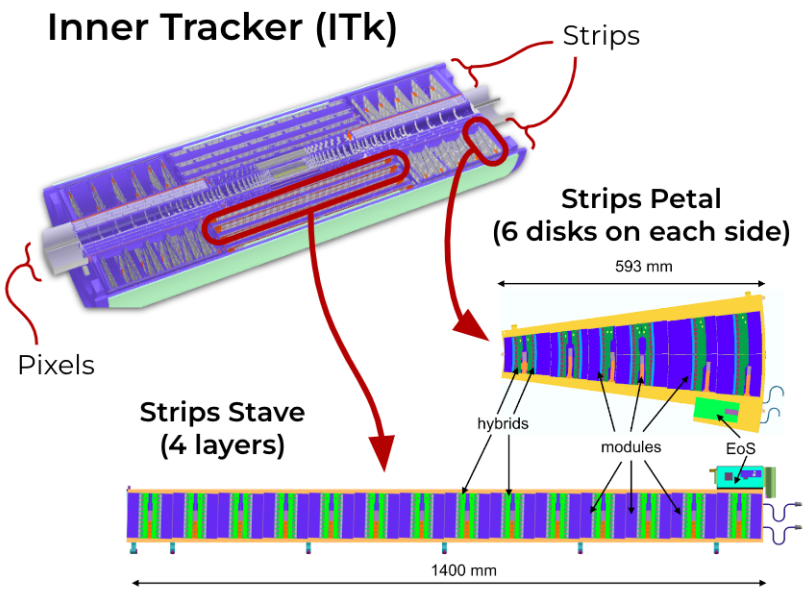

Figure 2: The Inner Tracker (ITk) and its main components. The ITk is composed of a Pixel Detector surrounded by a Strip Detector. The Strip Detector has four cylinders that form the barrel, composed of staves. Each stave hosts 14 modules on either side. The six disks on each side of the barrel are called the endcaps. These are composed of petals, each of which have six modules on either side. EoS: End of Substructure Card.

The exact anatomy of a module depends on its type. Endcap modules R3, R4, and R5 are split into two half-modules. Every (full) module has a silicon sensor, one or two "hybrids" (except for R3 modules, which has a total of four hybrids), and a powerboard [3]. Hybrids are printed circuit boards that host a varying number of application-specific integrated circuits (ASICs): these are the ATLAS Binary Chips (ABCs) and the Hybrid Controller Chips (HCCs). Important components of the powerboads are the DC-DC converter and the Autonomous Monitor and Control Chip (AMAC). In the case of the R3, R4, and R5 half-modules, only the left half-module contains HCCs while only the right half-module contains the powerboard. A detailed example of a module is shown in Figure 3 and a close look at a hybrid is shown in Figure 4.

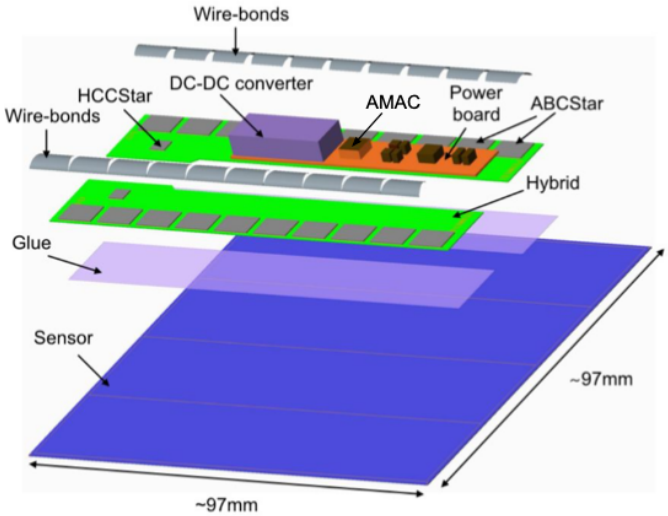

Figure 3: A short-strip (SS) barrel module. Like all modules, it is composed of a silicon sensor, a powerboard, and one or more hybrids. ABCStar: ATLAS Binary Chip, AMAC: Autonomous Monitor And Control Chip, HCCStar: Hybrid Controller Chip.

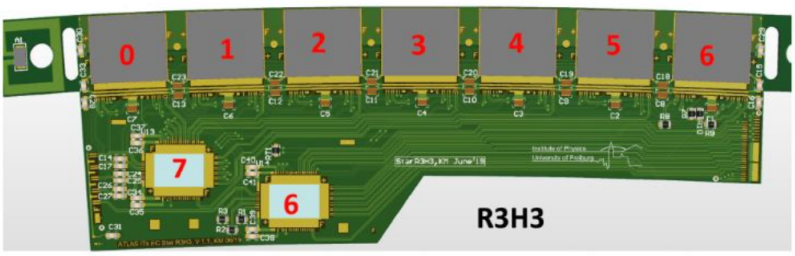

Figure 4: An R3H3 hybrid, one of four hybrids that belong on an R3 endcap module. This hybrid has seven ATLAS Binary Chips (ABCs) and two Hybrid Controller Chips (HCCs). Chips are labelled following a numbering scheme developed by the University of Toronto [4].

1.3 ITk development assigned to the University of Toronto

A large number of institutions are participating in the elaboration of the ITk, with different tasks assigned to each one. The University of Toronto is tasked with putting together all endcap hybrid types as well as endcap modules R0 and R3. For the production of these components, the University of Toronto has partnered with Celestica, a company that specializes in high-density electronics. Production and testing of these ITk parts will be mostly done at Celestica's Newmarket Microelectronics Lab, less than an hour's drive north of Toronto, Ontario.

2 Database scripts for the University of Toronto

2.1 The ITk Production Database and the Celestica spreadsheet

In order to keep track of all the moving pieces of ITk during production, the ATLAS group has mandated Unicorn University, in the Czech Republic, with developing and maintaining the ITk Production Database[1]. This Database keeps track of the location and status of components and tools, as well as test results and assembly of components. Unique identifiers are associated to both real objects (components, tools, receptacles, etc.) and virtual objects (dummy components, tests, composite objects, etc.). A common example of a composite object is the "hybrid assembly", which is the combination of a hybrid "flex" (the circuit board) and its assembled ASICs. Three major systems of identifiers exist in the Database: a 32-character hexadecimal "code" is used alongside a 24-character hexadecimal "ID" and—for most real components—a 14-character alphanumeric serial number. Other identifiers also exist, namely radiofrequency identification (RFID). RFIDs are recorded only for those components and tools that physically carry an RFID chip. Any of these identifiers can be used to look up an object registered in the Database, with varying simplicity.

Authorized users can access the Database through their CERN credentials or by inputting a unique combination of two passwords they have chosen (their two access codes).

Using the Database on an internet browser to register and assemble components or to upload test results is a slow and tedious endeavour. The University of Toronto and Celestica instead share a Google Spreadsheet where this same information is initially recorded. My task was to automate the transfer of specific information from the spreadsheet to the Database using Python scripts.

2.2 Transferring information from spreadsheet to database

A total of six task-specific scripts were written by my collaborator Ruchi Soni and myself. These were:

- *assemble_flex_to_assembly.py*, which assembles hybrid flexes to newly registered hybrid assemblies

[1] https://itkpd-test.unicorncollege.cz/

- *assemble_hybrid_to_module.py*, which assembles hybrid assemblies and powerboards to their corresponding modules
- *assemble_hybrid_to_panel.py*, which assembles hybrid assemblies to their corresponding test panels
- *hybrid_glue_weight.py*, which uploads results of glue weight tests to the corresponding hybrid assembly
- *module_glue_weight.py*, which uploads results of glue weight tests to the corresponding module
- *hybrid_wirebonds.py*, which uploads results of wire bonding tests to the corresponding hybrid assembly.

All the above scripts use the Python package *gspread* to communicate with the Google Spreadsheet and *requests* to send read and write requests to the Database. Since the Google Spreadsheet maintained by Celestica is private, scripts make use of the file *service-account.json*, which contains a private key associated with a special email address. This email address must appear in the list of accounts allowed to access the Google Spreadsheet. In our case, the scripts only need to be able to read the spreadsheet and do not make any modifications to it. For that reason the service account email address need only be listed as a Viewer—editing permissions are not necessary. Common functions used by these scripts are grouped in *helper_functions.py*. Other Python packages used are *typing*, *json*, *sys*, and *os*.

The Celestica spreadsheet is divided into several sheets, depending on the type of information contained. In the sheets that hold information that needs to be uploaded to the Database, each component is given its own row while the pertinent data are categorized into columns.

To use one of our scripts, one is meant to clone the GitLab repository where they are found (TorontoDatabaseScripts)[2] then access the directory "Glue Weights and Assembling" and run the Python script from a terminal. The user is expected to have their own access to the Celestica spreadsheet, for reference, but this is not crucial. However, the user must be able to access the Database through their two access codes. At the start, the user is asked to input the range of the rows they desire to process. To analyse the entire sheet, the user can safely input the range "0" to "end" ("end" being the only string accepted by the prompt). They are also asked for any exceptions in their range, that is, rows they don't want processed. The user is then asked to type their first and second Database access codes to begin the operation. As the program runs, it informs the user about its progression by printing progress messages. As it begins analyzing each row, the program prints the message, "Working on row x." During the process, the user is informed of each step that the program is taking. These messages can be, for example, "x successfully advanced from [stage A] to [stage B]," "x successfully disassembled from y," or "Test for x uploaded to ITk Database." When they arise, error messages returned by the Database are also transcribed to the user, often with added contextual explanations. Examples of these explanatory messages are "No component is found with the RFID x," or "Component x is not assembled to a parent." Programmed contextual messages like, "Please make sure the hybrid flex is assembled to a hybrid assembly," can also give insight on a problem. Modifications made to the Database through the script are recorded as having been done by the user.

In order to facilitate the task of transferring information from the spreadsheet to the Database, each upload and assembly script works with a text file that records the row numbers of complete and successful execution of the task. Each of these progress file is named identically to the corresponding script but with the suffix *_progress.txt*. The script writes the number of the working row on a new line in the text file if and only if it reaches completion for that row and the last response status code transmitted by the Database is a value in the 200s (which signifies that the request was successful). Thus, progress files are constructed automatically and locally. Progress can be shared with other users by pushing the textfile back to the GitLab repository. At the beginning of each script, before inputting any row exceptions, the user is prompted to indicate if they want to use the progress file to skip over any rows recorded as completed. (If the user chooses to not read the progress file, the script will nonetheless still write successful rows to the file.) Since it is a regular text file, a progress file can be

[2] https://gitlab.cern.ch/itktoronto/TorontoDatabaseScripts

modified very easily with a text editor. A progress file can be manually populated with rows and this will work as long as it is written one row per line.

2.3 Test upload scripts

This section will look in detail at the three test upload scripts. See Ruchi Soni's paper[3] for more information on assembly scripts.

Glue weight tests are, as the name implies, a measurement of the weight of the glue applied between assembled components. This measurement is usually done indirectly (subtracting the weight of the components from that of the final combination). The amount of glue needs to be sufficient to ensure the adhesion does not fail during the lifetime of the component but not so much that it overflows. If the weights calculated are not within certain limits, the test is marked as failed. The weight of the glue as well as that of the relevant components need to be uploaded to the Database. In the case of hybrid assemblies, the test evaluates the glue that binds the ASICs to the hybrid flex. This test is performed and must be recorded on the Database at the "ASIC Attachment" stage of the hybrid assembly, which is the initial stage for this composite object upon registration in the Database. As for modules, the glue weight test evaluates the glue that binds finished hybrids and powerboards (if applicable) to the module or half-module. This test is meant to be done and recorded at the "Glued" stage of the module, early in this component's development.

The wire bonding test is a careful assessment of the ASIC and hybrid wire bonds in a hybrid assembly. Each wire bond is labelled by a number depending on the hybrid type and according to a specific scheme [4]. The test involves recording the wire number of any problematic bond and the description of the problem. The generic descriptions used in the spreadsheet are one of four: "wire did not stick", "heel break", "lift", or "other defect". There is also a distinction to be made between those problematic wire bond that were successfully repaired and those that were not. This data must be uploaded to the Database at the "Wire Bonding" stage of the hybrid assembly, which is the stage that follows "ASIC Attachment".

For each row requested by the user, the test upload scripts gather the necessary information from the spreadsheet by reading the appropriate columns. The program does not search for the column, rather its location is hardcoded into the script. Some information needed by the Database is not found in a column of the spreadsheet but can be calculated from other values (e.g., the weight of the ASICs for *hybrid_glue_weight.py* or the total number of failed ASIC wire bonds for *hybrid_wirebonds.py*). The pass or fail status of the test is explicitly found in a column for *hybrid_glue_weight.py*, while it must be deduced for the other two. In *module_glue_weight.py*, if any glue weight is too far from a hardcoded ideal value for its category, the test is recorded as failed for that component. In *hybrid_wirebonds.py*, the test is recorded as failed if there is at least one problematic wire bond that could not be repaired. In all three scripts, the component in question in each row is found through the RFID of the hybrid flex associated with the hybrid assembly (and, in turn, associated with the module, if applicable).

Unlike the other two test upload scripts, *hybrid_wirebonds.py* verifies that the component is at the proper stage before proceeding with the test upload. This is because the wire bonding test marks the time at which the hybrid assembly stage must be promoted to "Wire Bonding". If the hybrid assembly is at the previous stage, the script will attempt to upstage it before continuing. The Database will not accept the wire bonding test upload at any other stage.

When given a complete test upload request, the Database will not check to see if this upload had already been done beforehand. There is therefore a risk of duplicating test uploads. To avoid this, all three test upload scripts use a run number as a means of identifying the test. By default, the run number is set to "1" in all three scripts. Before uploading, the script sends a read request to search the Database for other tests of the same type for the component in question with the same run number ("1"). If at least one such test is found, it is assumed to be the same test and the script will

[3] https://particlephysics.ca/research-activities/undergrad-program-cern-papers/?lang=en

not automatically upload. It is important to note that the scripts do not actually read and compare the individual values of the tests to determine if it is truly a duplication—this assumption is based only on the run number. For this reason, in such a situation, the user is solicited to input "yes" to continue with the upload or "no" to abort it. Figure 5 shows excerpts from *helper_functions.py* that are responsible for this checking.

```python
def upload_to_ITkDB(d: dict, headers=None, force_upload=False) -> bool:

    """Upload test results in d to ITk Databse. If headers is None, user
    authentication is necessary to obtain token and headers. If force_upload is
    True, upload the test even if there is another test in the database for this
    component with the same run number, otherwise the duplicate test is not
    uploaded.  Return True if upload is successful, otherwise return False.
    """

    if headers == None:
        headers = get_headers()

    if is_duplicate(d, headers) and not force_upload:
        print('{} test with run number {} already exists'.format(
                  d['testType'], d['runNumber'])
              + ' for {} in ITk Database.'.format(d['component']))

        answer = None
        while type(answer) != bool:
            answer_str = required_input(
                '\nWould you like to upload this test anyway?'
                + ' (Enter Y or N)\n'
            )
            answer = convert_to_bool(answer_str)

        if not answer:
            print('Test upload aborted for {}.'.format(d['component']))
            return True


    json_str = json.dumps(d, indent=4)

    json_bytes = bytes(json_str,'utf-8')
    r = requests.post(url=URL_POST_TEST, data=json_bytes, headers=headers)
```

(a)

```python
def is_duplicate(d: dict, headers=None) -> bool:

    """Return True if and only if  there already exists a test in the
    ITk Database with the same run number as d for the test type and component
    in question. If headers is None, user authentication is necessary to obtain
    token and headers.
    """

    (r_code, info_dict) = get_info_response(sn=d['component'], headers=headers)

    duplicate = False

    if 'tests' in info_dict:

        # info_dict['tests'] is a list of test descriptions for this component
        # where each test description is presented as a dictionary
        for test_type in info_dict['tests']:

            # Look for the presence of a test with the same test code
            # in info_dict
            if test_type['code'] == d['testType']:

                # Look for the record of completed tests with this test code
                # for this component in info_dict
                if 'testRuns' in test_type:

                    # If a completed test in info_dict has the same run number
                    # as the test represented by in d,
                    # then consider the test in d to be a duplicate
                    for test_run in test_type['testRuns']:
                        if test_run['runNumber'] == d['runNumber']:
                            duplicate = True

    return duplicate
```

(b)

Figure 5: (a) First portion of the function "upload_to_ITkDB" in *helper_functions.py*. The function is designed to ask the user if they want to upload a possibly duplicate test to the ITk Production Database. (b) The entire function "is_duplicate" in *helper_functions.py*, which looks for test uploads with an identical run number for the same component in the Database.

2.4 Additional functionalities of the scripts

The Python scripts support a number of modifications to alter their performance. Many constants defined at the beginning of the scripts can be changed if needed. Such changes may be necessary to ensure the continued performance of the scripts when changes are made to the spreadsheet or perhaps to accepted values of glue weight tests.

If any alteration is made to how the Celestica spreadsheet is organized, it is crucial to redefine the column mapping. This map is simply the constants that represent the relevant data columns defined at the onset of every script. It is important to note that the assigned value to these constants is the column's index from 0, so the nth column in the spreadsheet has value $n-1$ assigned to its constant. The program is hardcoded to take information from the right column in this way.

Since the script *module_glue_weight.py* analyzes the success of the test automatically to determine if the component passes or fails, a change in acceptable glue wieght standards must be accompanied by a change of the constant Python dictionaries "GW_IDEALS" and "GW_TOLERANCES". These constants represent respectively the values for the ideal glue weights per component type and those for the accepted tolerance for any deviation from the ideal, both in milligrams.

One rather trivial change that can be made to any test upload script is redefining the constant "DEFAULT_RUN_NUMBER" (originally equal to "1"). This can be changed to any string and is used by the program to help individualize test uploads in order to minimize accidental duplications.

The scripts also include a number of functionalities that are not utilized under normal operation. These can be harnessed by changing arguments in functions written in the test upload scripts or in *helper_functions.py*.

In the case that a user may want to intentionally upload successive tests of the same kind to the same component, the user can enter their own run number for every test upload. To do so, the argument for the parameter "run_number" from "DEFAULT_RUN_NUMBER" to "None" in the functions "get_hybrid_gw_test_results", "get_module_gw_test_results", or "get_hybrid_wb_test_results" in their respective test script. As a result, the user will be prompted to enter a run number of their choice for each test upload during the execution of the program. Otherwise, it is possible to make the program simply forgo any checking for duplicates. This means the program uploads any complete test to the Database regardless of the existence of another test for the same component with an identical run number. This is done by changing the argument for the parameter "force_upload" from "False" to "True" in the function "upload_to_ITkDB".

Another modification involves adding the ability to tag a test result as problematic. The Database is designed to allow the user to flag whether or not there was a problem in the execution of a test. However, as of this writing, the Celestica spreadsheet has no area to indicate this possibility and so it is not expected that a test should be flagged as problematic during automated uploads via the Python scripts. Nonetheless, the test scripts are designed to allow such flagging after changing the argument for the parameter "problem" from "False" to "None" in the specific functions used to construct the Python dictionary that will be sent to the Database ("get_hybrid_gw_test_results", "get_module_gw_test_results", and "get_hybrid_wb_test_results"). Doing so will trigger a prompt for each test upload during the execution of the program asking the user to input "yes" or "no" to indicate the presence of a problem. Changing this argument to "True" instead automatically flags all uploaded tests as problematic. (This likely has limited usefulness.)

The function "save_as_json" defined in *helper_functions.py* is originally unused. It can be added to any of the test scripts just before the function "upload_to_ITkDB" to save the test results gathered from the spreadsheet as a json file on the user's local machine.

Finally, many functions defined in *helper_functions.py* can also be used manually to read information from the Database much faster than can be obtained by navigating using a browser. A user would simply need to open a Python shell, import *helper_functions.py* and call on functions such as "get_rfid_from_sn", which returns the RFID of a component given another unique identifier, or "get_component_subtype", which returns the subtype of a component given its identifier, or "get_child_codes", which returns a list of the component codes of all the components assembled as "children" (subordinates) to a given component identified by a unique identifier.

3 Discussion

3.1 Strengths of the scripts

The biggest strength of these database scripts is that they resist crashing when they encounter a problem or error. Instead of quitting, the program will abandon the current working row and move to the next row in the requested range. Some errors can arise from incomplete or inconsistent information in the spreadsheet; the program can often identify these and announce it to the user through a printed message (e.g., "No RFID given to search for," or "Wirebond number not in U of T catalog") before skipping to the next row. Other, more insidious errors can arise from the Database refusing a request for one of a multitude of reasons (e.g., the component cannot be found, the component is in the wrong location, the component is at the wrong stage, etc.). When a row is skipped, it is not recorded as completed in the corresponding progress text file, even if some steps of the process had been successful.

All scripts are designed with the idea that a user may intentionally or accidentally process rows that have already been done, in part or in full. The scripts are designed to not duplicate any work on the Database. Assembly scripts will verify the current assembly status and component stage before sending a write request to the Database. Since these assembly scripts usually send a sequence of many write requests, a verification is done before every such request—the program does not assume that if a previous step is complete then subsequent steps are also complete. Test upload scripts will search for the existence of a test with an identical run number before sending the upload.

Generally, relying on a user for inputs during the execution of a program can lead to unexpected inputs. For this reason, the database scripts anticipate that it might receive nonsense answers from the user and are designed to react appropriately to the wrong type of input. For example, when asking a yes/no question, the program will accept (without regard to letter case) "yes", "y", "true", and "t" as yes and "no", "n", "false", and "f" as no; all other answers will prompt the question again. When asked for a range of rows, the program will only accept integers (and for the endpoint, it will also accept the string "end"), prompting again otherwise. If the range given is larger than what exists in the spreadsheet (e.g., if asked for rows -3 to 5000), the program will still analyze any existing rows within this range. When asking what rows the user wants to skip, any nonsense or out-of-bound answers are ignored. The excerpt from *helper_functions.py* that allows for this robust management of user input is shown in Figure 6.

```
153 def get_rows_from_user(data):
154     '''Asks the user for the rows where they would like to operate, which may
155     depend on data.'''
156 
157 
158     start_row = input(
159         "Enter the row number where you would like to begin processing.\n"
160     )
161 
162     while type(start_row) != int:
163         try:
164             start_row = int(start_row)
165 
166         except:
167             print('Invalid input. Input must be an integer.')
168             start_row = input(
169                 'Enter the row number where'
170                 + ' you would like to begin processing.\n'
171             )
172 
173     end_row = input(
174         'Enter the row number where you would like to end processing.\n'
175         + 'To go until the end, type "end".\n'
176     )
177 
178     while type(end_row) != int:
179 
180         if end_row.lower() == "end":
181             end_row = len(data)
182 
183         else:
184             try:
185                 end_row = int(end_row)
```

Figure 6: First portion of the function "get_rows_from_user" in *helper_functions.py*. The function prompts the user on a loop until it receives a reasonable input.

Another problem foreseen in the scripts is the possibility of the user's access expiring in the middle of the execution of the program. This is unlikely to happen during normal usage as this would require a considerably large number of rows to analyze in order for credentials to expire after over a dozen minutes. If this happens, however, the user is warned and prompted to input their two access codes again to re-authenticate. When doing so, no progress is lost.

3.2 Limitations of the scripts

A few events will in fact make the programs quit. One quick way to end one of these programs is to input the wrong access codes. Another possible reason for the program to quit is when it encounters a database timeout (if the Database is down or unresponsive). Other unpredictable database errors can make the operation stop. Fortunately, the progress of successful rows made before an error or interruption is still recorded in the appropriate progress file. Because of the erratic nature of these errors and interruptions, the best course of action when a script fails for these reasons is to simply run the script again.

A major weakness of the database scripts is their high sensitivity to changes in the structure of the Celestica spreadsheet. Unless specific edits are made to the scripts, they will tolerate no sheet name changes and no rearrangement of columns. If rows are rearranged after an initial usage of a script, any mixing of rows in the spreadsheet may result in the loss of coordination with the corresponding progress file. Inconsistency in the date format used in the spreadsheet may also cause errors: the programs are designed assuming all dates remain in the format DD/MM/YYYY or DD-MM-YYYY.

The scripts can only be as good as the Celestica spreadsheet allows. Any errors in the spreadsheet may ultimately be sent to the Database. If the error is nonsensical enough (e.g., incompatible components to assemble), the Database will fortunately refuse the write request and subsequently return the corresponding error message. However, some sneaky errors such as an incorrectly recorded glue weight will not be refused by the Database and that error will be uploaded. The scripts are not designed—and do not have the permissions—to correct anything in the spreadsheet.

3.3 Challenges

One major challenge in developing code for the ITk Production Database was that it is overly fastidious. A write request will be refused by the Database for a variety of trivial reasons, often related to the component's status, stage, or location not being up-to-date on the Database. Often times, these updates need to be brought by other collaborators from other institutions or may require the intervention of a user with high authority in the Database. A lot of coding was dedicated to working with such obstacles, which were discovered after considerable trial and error.

The Database will occasionally fail to recognize a component by its serial number during certain action requests, but it will still recognize the component code or ID. This happened very rarely—we only saw this with some assembly requests. To work around this oddity, the scripts are equipped to look up a different identifier for the component when an action request fails with a serial number. The irony in this situation is, of course, that the serial number is successfully used to find another identifier but not to perform an action such as assembling.

The Celestica spreadsheet was also a source of challenges. Scripts had to be reworked during their development following last-minute structural changes in the spreadsheet. Most dates were written in the DD/MM/YYYY format, but the MM/DD/YYYY was occasionally used, providing the wrong date to the Database. A lot of work was done to have the programs try and recognize problems in the spreadsheet before attempting to send a write request to the Database.

3.4 Impact

In the few weeks since their development, the six database scripts developed by Ruchi Soni and myself have already proven useful to the team at the University of Toronto. Some sections of our code are being used in future scripts aimed to perform other tasks with the Database.

4 Conclusions

In summary, six Python scripts were written with the objective of uploading data from the Google Spreadsheet shared between Celestica and the University of Toronto to the ITk Production Database. Three scripts sent information on component assembling and three scripts sent component test results.

All scripts keep the user informed of the steps they are taking during their run and resist crashing when they encounter one of many common problems in an upload. The scripts are capable of recognizing some problems and communicating them to the user. They keep track of their progress locally in a text file, which is easy to modify and push back to the remote repository if desired. The written programs are, however, highly sensitive to any structural changes or content errors in the spreadsheet.

The Python scripts show promise in their usefulness to teams at Celestica and at the University of Toronto working on the ITk upgrade. They are on track to being used extensively once component production begins in earnest. Future scripts for uploading other information to the ITk Production Database may be developed using sections and ideas from these ones.

References

[1] The ATLAS Collaboration, "Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lhc," *Physics Letters B*, vol. 716, no. 1, pp. 1–29, 2012.

[2] The CMS Collaboration, "Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC," *Physics Letters B*, vol. 716, pp. 30–61, sep 2012.

[3] The ATLAS Collaboration, *Technical Design Report for the ATLAS Inner Tracker Strip Detector*. CERN, 2017.

[4] W. Trischuk, L. Valoce, D. Sperlich, K. Dette, S. Beaupre, L. Poley, S. Issinski, and C. S. Contell, *Endcap Hybrid Wire-Bond Documentation: STAR Chipset V2.1*. 2022.