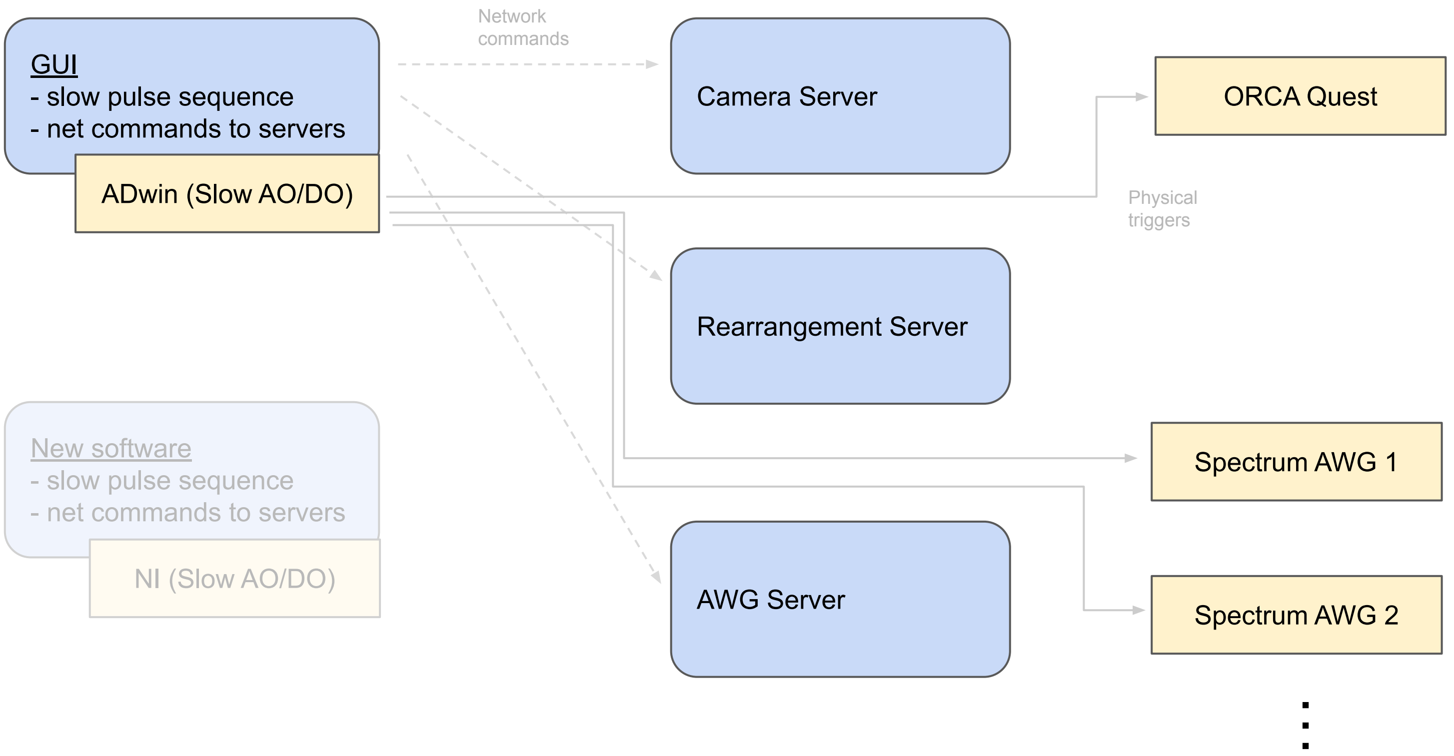
NI Experiment Control System

Agenda

1. Control system overview: what we’re changing
2. Path to our current solution
3. Detour on synchronization
4. Current solution: features and front end
5. Current solution: back-end

Overall experiment control system suggestion -
- copy everything from AA1, only replace ADwin + old software



LabScript limitations

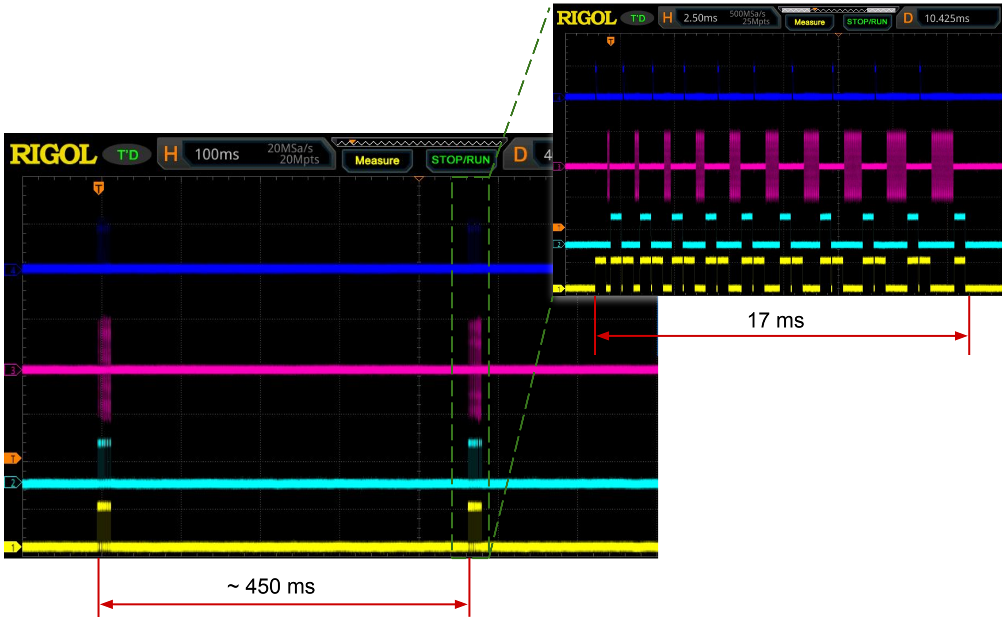
LabScript - a good hybrid:

- script-based sequence definition
- GUI control of sequence params

Limitations: script loading takes >450ms

- If only one physical repetition per script - max 2 Hz rep rate

- Also dependent on signal length

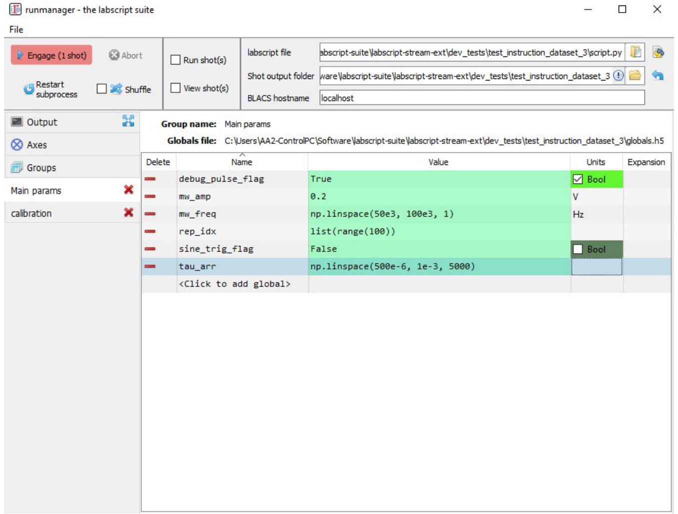


Ways to address:

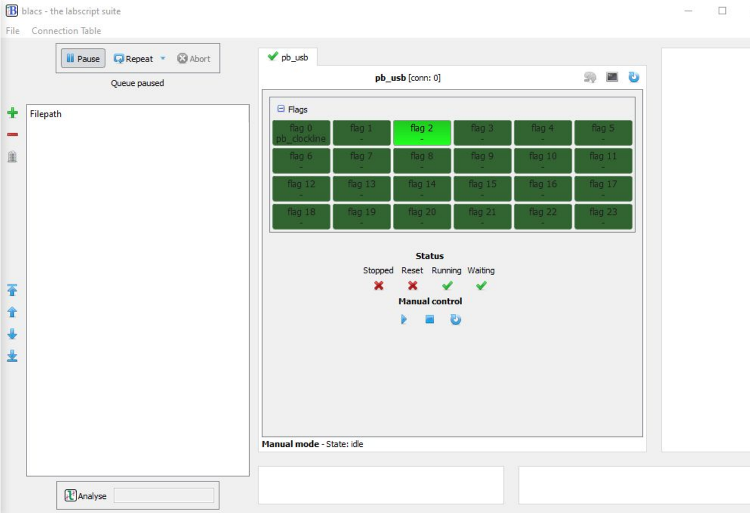
- Eliminate signal-length dependence
- Decrease script-load overhead

LabScript overview

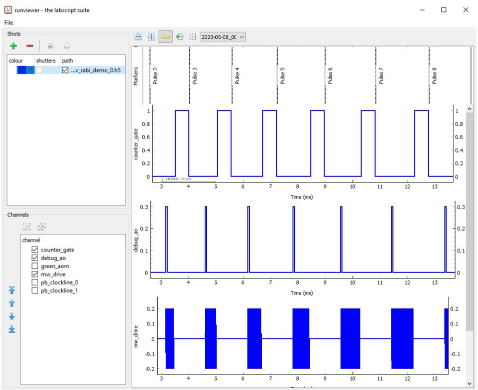
```
t = 0
ls.add_time_marker(t, "Start", verbose=True)
ls.start()

for idx, tau in enumerate(tau_arr):

    ls.add_time_marker(t, f"Pulse {idx}", verbose=True)

    if debug_pulse_flag:
        debug_ao.constant(t=t, value=1.5)
        debug_ao.constant(t=t+50e-6, value=0)

    # Init pulse
    green_aom.go_high(t=t)
    t += 500e-6
    green_aom.go_low(t=t)

    # Rabi
    t += 50e-6
    mw_drive.sine(
        t=t,
        duration=tau,
        amplitude=mw_amp,
        angfreq=2*np.pi * mw_freq,
        phase=0,
        dc_offset=0,
        samplerate=1e6
    )
    t += tau
    mw_drive.constant(t=t, value=0.0)

    # Readout pulse
    t += 50e-6
    green_aom.go_high(t=t)
    counter_gate.go_high(t=t)

    t += 500e-6
    green_aom.go_low(t=t)
    counter_gate.go_low(t=t)

    t += 50e-6

t += 500e-6
ls.stop(t)
```



RunManager GUI

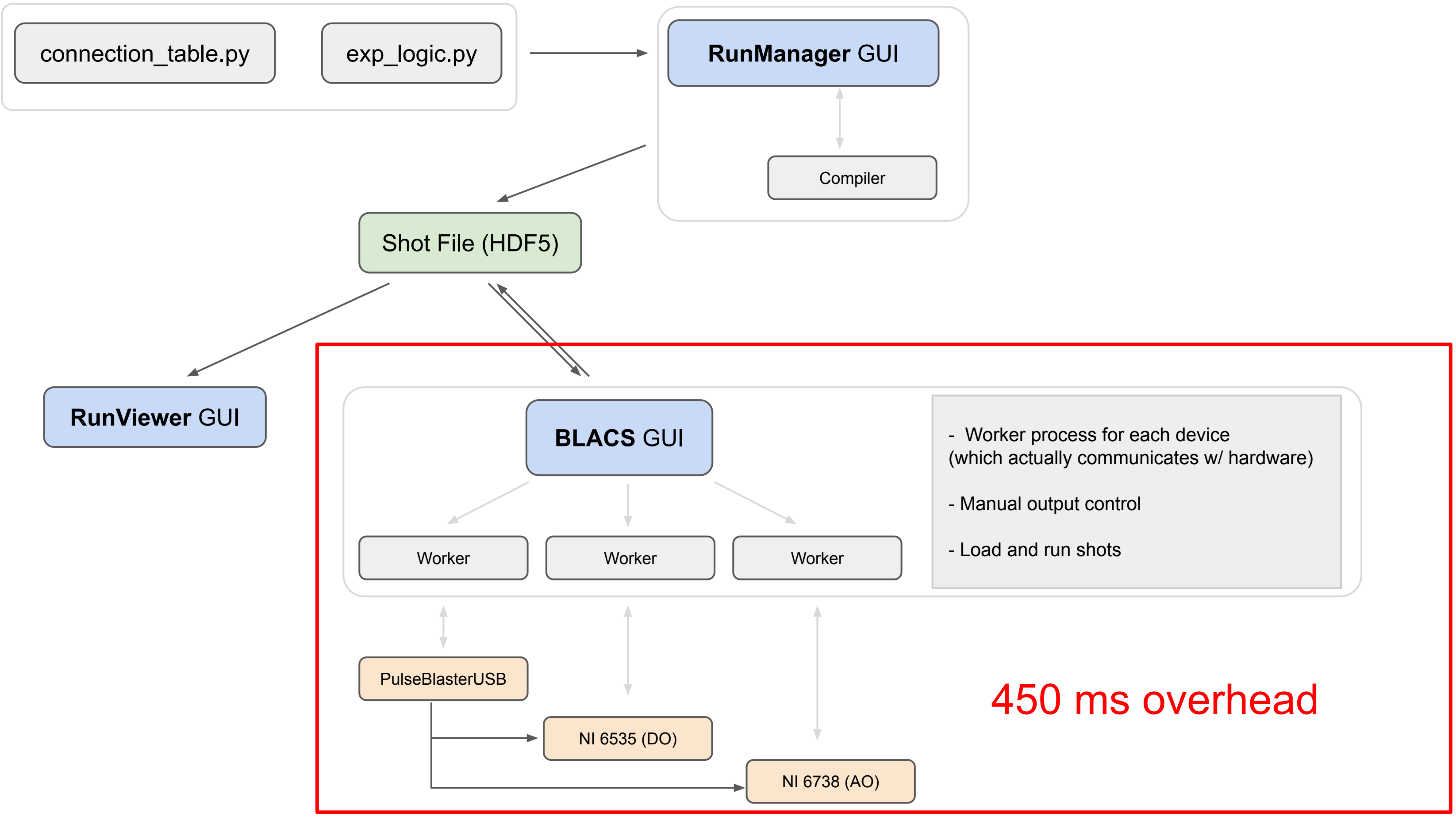


BLACS GUI



RunViewer GUI

Typical LabScript workflow / main components



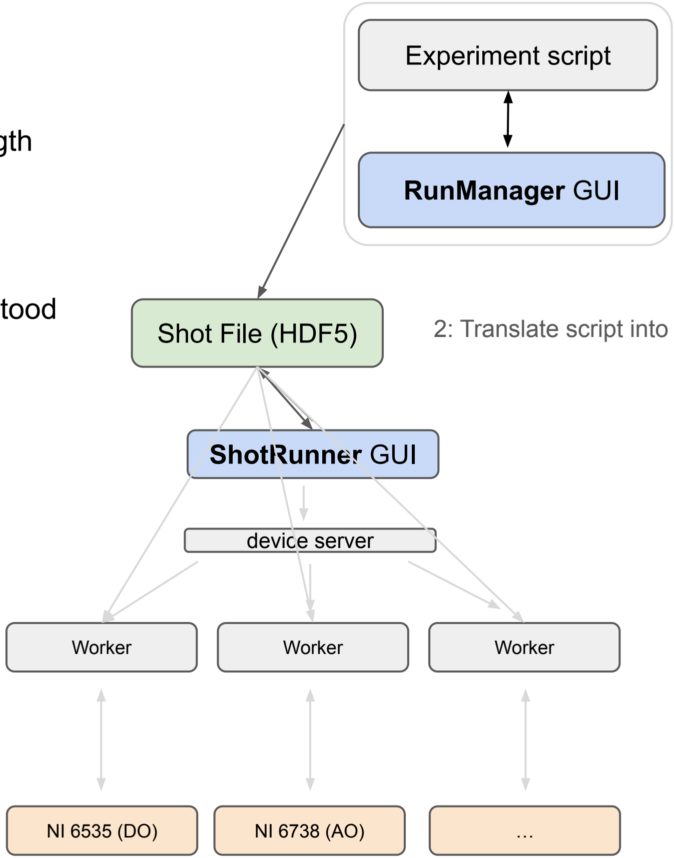
First modification (7.26): custom streaming shot-runner

Improvements:

- Runner overhead 450ms to 120ms
- Eliminated dependence on signal length

Problems:

- Labscript compiling is not fully understood
- GUI potentially redundant
- Complex server-client development, debugging, and cleanup



1: Write python code in compiler standards to specify the experiment design

2: Translate script into shotfile

3: Start the shotrunner client and server (if not running already)

4: Add .hdf5 files of interest to running queue (modify #repetitions for each file, if needed)

5: Ask the shotrunner to execute the queue

Second modification (8.16): full-stack Python

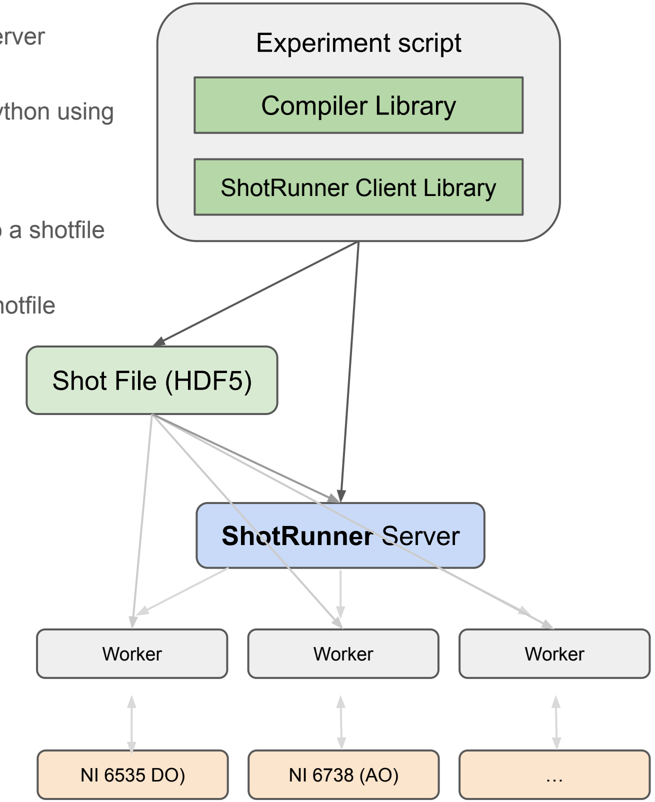
Improvements:

- Full-stack control over compiler behavior
- Compilation and running are easily automated

Problems:

- Unnecessary shotfile
  - Each worker essentially “reconstructs” the experiment
- Complex server-client development, debugging, and cleanup
  - Zombie workers after server crash
  - Unresponsive server after worker crash

1. (Re)start the ShotRunner server
2. Design the experiment in Python using compiler library
3. Compile the experiment into a shotfile
4. Ask the server to execute shotfile



Third modification (9.7): Full streamline with Rust backend

Benefits

- Eliminate shotfile overhead (-40ms in the loop)
- One experiment, one script, one process
- Remove Python performance limitations

Non-Pythonic-backend: Why?

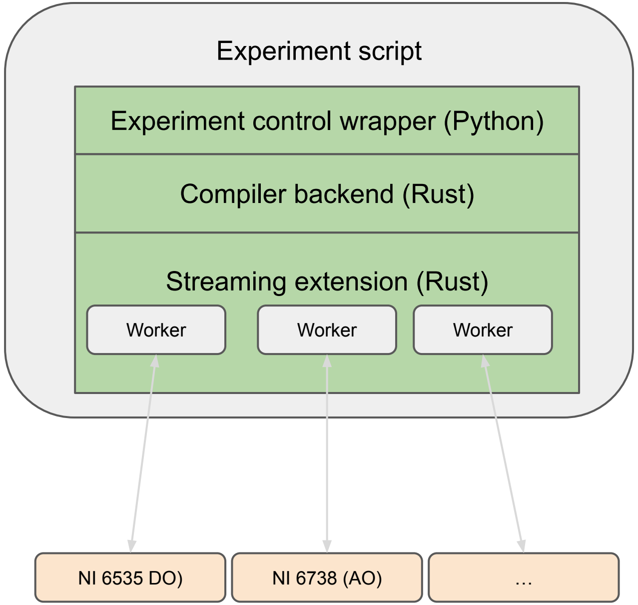
- Circumvent the Python GIL
- Multi-threading (~1ms) v.s. Python multiprocessing (~200ms)
- Stronger compiler checks means less buggy code

Why Rust?

- Interfaces easily with Python (front end) and C (NI driver)
- Very strong safety guarantees: less bugs
- Friendly development pipeline

Trade-off

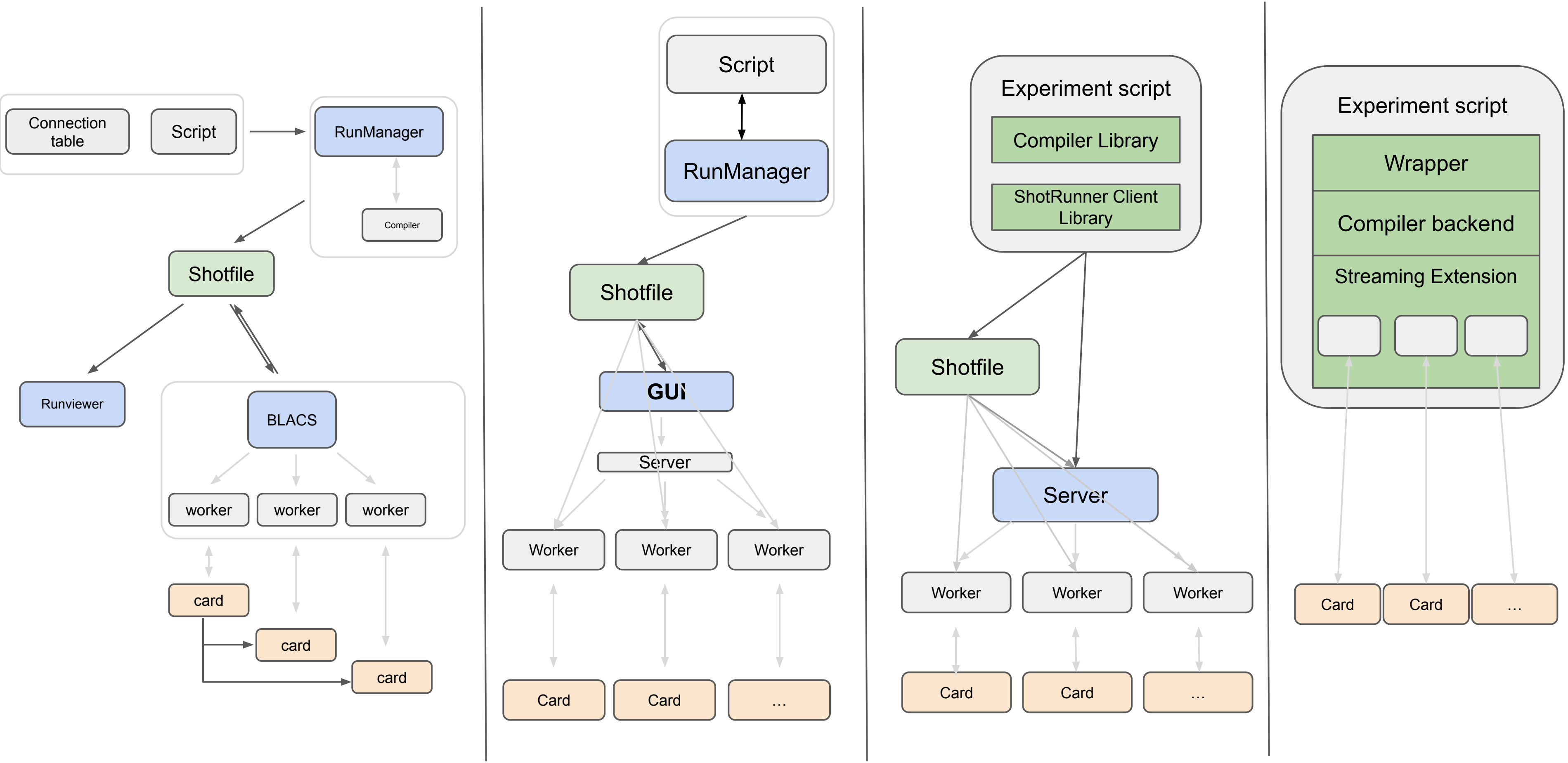
- Modifying streaming behavior requires editing Rust source



1 . Design the experiment in Python using the experiment control library.

2. Stream the experiment

Progress Recap: streaming and streamline



Detour: Synchronization

Method	Description	Pros and Cons
Start Trigger Synchronization	Devices share a common start trigger	● Most flexible, synchronizes between tasks and devices with different rates ● Inter-device drift increases with task length
Share Sample Clock	Devices obtain their sample clock from a designated source	● Supported on all devices ● Tasks need to run at the source rate
Reference Clock Synchronization	Devices phase-lock their clocks with a common reference	● Flexible, synchronizes between tasks and devices with different rates ● Not supported on all devices

Detour: Synchronization

Primary AO

- Exports start trigger (PXI_Trig0)
- Export 10MHz reference (PXI_Trig7)
- Flexible sampling rate

Secondary AO

- Expects start trigger (PXI_Trig0)
- Reference-locks to external 10MHz reference (PXI_Trig7)
- Flexible sampling rate

Secondary DO

- Expects start trigger (PXI_Trig0)
- Imports sampling clock from 10MHz reference (PXI_Trig7)
- Fixed sampling rate: 10MHz