GLIDER: Comprehensive Design Specification for Modular Experimental Orchestration

1. Introduction and System Philosophy

The contemporary laboratory environment is increasingly defined by the integration of heterogeneous hardware systems, ranging from commodity microcontrollers like Arduino and Raspberry Pi to specialized industrial actuators and sensors. While Python has established itself as the lingua franca of scientific computing due to its rich ecosystem of data analysis libraries, the domain of experimental orchestration—specifically the real-time control of hardware coupled with complex logic flows—remains fragmented. Researchers often resort to monolithic, brittle scripts that are difficult to modify, impossible to scale, and challenging to debug. The GLIDER (General Laboratory Interface for Design, Experimentation, and Recording) project aims to address this systemic inefficiency by providing a robust, open-source software architecture that prioritizes modularity, scalability, and user accessibility.

This design document articulates the architectural vision for GLIDER, a system requirements-compliant application capable of operating across a spectrum of computing environments, from high-performance desktop workstations to resource-constrained embedded systems like the Raspberry Pi. The core philosophy driving GLIDER's design is the strict separation of concerns between the logical definition of an experiment, the physical actuation of hardware, and the graphical presentation of state. This separation allows the software to serve two distinct but interrelated roles: a sophisticated Integrated Development Environment (IDE) for constructing experimental flows, and a streamlined, touch-optimized runtime interface for execution in the field.

The analysis of current Python GUI frameworks and hardware abstraction methodologies suggests that a monolithic approach is insufficient for these requirements. Instead, GLIDER adopts a split-core architecture utilizing **PyQt6** for the graphical shell, **Ryven** for the underlying flow-based logic engine, and **asyncio** (integrated via **qasync**) for managing the high-concurrency demands of hardware I/O without blocking the user interface. This document details the technical specifications, design patterns, and implementation strategies required to realize this vision, ensuring that GLIDER is not merely a script runner, but a scalable platform for experimental science.

1.1 The Role of Python in Embedded Orchestration

Python's dominance in the embedded sector has grown significantly, transitioning from a

scripting language to a primary driver for hardware control.[1] The availability of libraries such as RPi.GPIO and gpiozero has lowered the barrier to entry for physical computing.[2] However, the direct utilization of these libraries within a graphical application often leads to "spaghetti code" where UI logic is tightly coupled with hardware timing, resulting in interfaces that freeze during sensor reads or motor movements.

GLIDER addresses this by treating Python not just as the glue code, but as the hosting environment for a sophisticated event loop that arbitrates between the user's intent and the hardware's capabilities. By leveraging the advanced features of Python 3.8+, specifically Abstract Base Classes (ABCs) and asynchronous coroutines, GLIDER ensures that hardware drivers are swappable plugins rather than hardcoded dependencies.[4] This allows a researcher to prototype an experiment using an Arduino Uno and seamlessly migrate to a robust industrial controller by simply changing the driver selection in the software, preserving the high-level logic flow.

1.2 Addressing the Dual-Interface Requirement

A unique constraint of the GLIDER specification is the necessity to support both a standard desktop environment and a 480x800 vertical touchscreen interface on a Raspberry Pi. This requirement fundamentally shapes the UI/UX architecture. Desktop interfaces benefit from high information density, multi-window workflows, and precise mouse interaction. In contrast, a vertical touch interface on a Raspberry Pi (often used in "kiosk mode") demands large hit targets, simplified navigation hierarchies, and high-contrast visuals for readability in lab environments.[5]

The design proposed herein utilizes a responsive view management strategy. The core application logic remains invariant, but the presentation layer creates distinct view hierarchies based on the detected runtime environment. This avoids the maintenance burden of developing two separate applications while ensuring that the "Runner" experience on the Pi is not merely a shrunken version of the Desktop "Builder," but a purpose-built interface for experimental execution.

---

2. Architectural Overview and Design Patterns

The architecture of GLIDER is predicated on the Model-View-Controller (MVC) pattern, adapted for asynchronous event-driven programming. This structure is essential to decouple the high-frequency events of hardware sensors from the rendering cycle of the Graphical User Interface (GUI).

2.1 The Split-Core Architecture

To ensure scalability and testability, GLIDER comprises two primary subsystems: the **GLIDER**

Core and the **GLIDER Shell**.

The **GLIDER Core** is a headless, pure-Python library responsible for the lifecycle management of experiments. It loads hardware drivers, maintains the state of the "Flow" (the logical graph of the experiment), manages data logging, and executes the event loop. Crucially, the Core has no dependencies on PyQt6 or any display libraries. This design decision ensures that the Core can be run as a system service (daemon) on the Raspberry Pi, executing experiments autonomously even if the GUI process terminates or if the device is running in a headless configuration.[7]

The **GLIDER Shell** is the PyQt6 application that provides the visual interface. It acts as the "View" and "Controller" in the MVC paradigm. The Shell communicates with the Core via a strict API facade. When a user drags a node onto the canvas in the Shell, a command is sent to the Core to instantiate the corresponding logic object. When the Core receives new sensor data, it emits a signal that the Shell consumes to update a label or plot. This decoupling is vital for stability; an unhandled exception in the GUI rendering logic should not crash the ongoing physical experiment managed by the Core.

2.2 Concurrency Model: The Event Loop

Traditional hardware control software often relies on multi-threading to handle simultaneous tasks (e.g., reading a sensor while spinning a motor). However, Python's Global Interpreter Lock (GIL) limits true parallelism, and managing thread safety—avoiding race conditions and deadlocks—adds significant complexity.[8] Furthermore, hardware operations are typically I/O-bound (waiting for a serial response) rather than CPU-bound, making them ideal candidates for cooperative multitasking.

GLIDER employs **Python's asyncio** library as its primary concurrency model. By defining hardware interactions as non-blocking coroutines (async def), the system can handle hundreds of simultaneous device inputs on a single thread without blocking. To integrate this with the PyQt6 GUI, which has its own event loop, GLIDER utilizes the **qasync** library.[9] qasync is an implementation of the asyncio event loop that runs on top of the Qt event loop. This allows Qt signals/slots and asyncio futures/tasks to coexist seamlessly. A button click in PyQt can trigger an async function that communicates with an Arduino, waits for a response, and updates the UI, all without freezing the interface.[11]

2.3 Modularity and Plugin System

To satisfy the requirement for custom boards and devices, GLIDER implements a strict plugin architecture. The Core does not contain hardcoded support for specific hardware. Instead, it defines **Abstract Base Classes (ABCs)** that specify the contract a driver must fulfill (e.g., write_digital, read_analog).

Plugins are Python packages managed via importlib and importlib.metadata (formerly

pkg_resources). At startup, the GLIDER Plugin Manager scans a designated directory (e.g., ~/.glider/plugins) and the Python path for installed packages advertising the glider.driver entry point.[12] This allows researchers to distribute drivers for their custom hardware (e.g., a "Custom Atomizer Controller") as standalone Python packages that GLIDER automatically detects and integrates into the UI.

2.4 Data Flow vs. Execution Flow

Inspired by visual programming environments like Unreal Engine Blueprints and the underlying mechanics of Ryven, GLIDER distinguishes between two types of signal propagation within the experiment graph [13]:

Data Flow: This represents the continuous propagation of state. When a sensor node updates its value, that data flows downstream to any connected nodes (e.g., a "Threshold" check node). This is reactive programming; the graph re-evaluates automatically when inputs change.

Execution Flow: This represents the imperative sequence of actions. Nodes can have "Exec" inputs and outputs (visualized as distinct white lines). An "Action" node (e.g., "Start Motor") performs its task only when it receives an Execution signal, regardless of the state of its data inputs. This distinction is critical for defining experimental protocols (e.g., "Wait 10 seconds, THEN turn on the pump") which cannot be modeled purely with reactive data flow.[14]

---

3. Graphical User Interface (GUI) Strategy

The selection of the GUI framework is a pivotal design decision. The requirements entail a sophisticated node-graph editor for the creation phase and a touch-optimized dashboard for the execution phase.

3.1 Framework Selection: PyQt6

While frameworks like **Kivy** are often recommended for touch interfaces due to their mobile-first design [15], they lack the mature desktop widget ecosystem required for a complex IDE-like interface. **Tkinter** is insufficient for modern, high-performance graphics and lacks built-in support for complex custom widgets like node graphs.[16]

PyQt6 (and the underlying Qt 6 framework) is the optimal choice for GLIDER for several reasons:

1. **The Graphics View Framework:** Qt provides QGraphicsScene and QGraphicsView, a highly optimized 2D rendering engine capable of handling thousands of interactive items (nodes, wires, ports). This allows GLIDER to implement a professional-grade visual scripting environment that remains performant even on the Raspberry Pi.[17]

2. **Hardware Acceleration:** Qt 6 leverages the GPU for rendering (via RHI - Rendering Hardware Interface). On the Raspberry Pi 4 and 5, this ensures that the UI renders at a smooth 60 FPS, which is critical for touch responsiveness.[19]
3. **Stylesheets (QSS):** Qt Style Sheets allow the application's appearance to be radically transformed using a CSS-like syntax. This enables GLIDER to switch between a "Desktop Theme" (compact, mouse-oriented) and a "Touch Theme" (large buttons, high contrast) dynamically without changing the underlying C++ or Python logic.[20]
4. **Cross-Platform Consistency:** PyQt6 ensures identical behavior on Windows, macOS, Linux, and Raspberry Pi OS, fulfilling the requirement for portability.[22]

3.2 Responsive Interface Design

To accommodate the 480x800 vertical resolution of the Raspberry Pi screen, GLIDER employs a ViewManager that detects the display properties at startup.

Table 1: UI Adaptation Strategy

Feature	Desktop Mode (PC)	Runner Mode (Raspberry Pi 480x800)
Window Layout	QMainWindow with Dockable Widgets (QDockWidget) for Library, Properties, and Console.	Single-window QStackedLayout or full-screen QWidget. No floating docks.
Navigation	Menu bars, detailed toolbars, keyboard shortcuts.	Large distinct tabs or a "Hamburger" side menu. Swipe gestures.
Node Graph	Full interactive editing: pan, zoom, drag-and-drop, context menus.	Read-only visualization or completely hidden. Focus is on dashboard controls.
Input Widgets	Standard QSpinBox, QLineEdit, small checkboxes.	Custom "Touch Widgets": Large toggle switches, dial knobs, virtual numeric keypad.
Scrolling	Mouse wheel, thin scrollbars.	Kinetic scrolling (QScroller), wide touch-friendly scrollbars.

The application detects the screen resolution via QScreen.size(). If the width is detected as ≤ 480 pixels, the application automatically launches in "Runner Mode," loading the specific QSS stylesheet that overrides widget padding, font sizes, and touch target dimensions.[6]

3.3 The Node Graph Editor

The visual scripting interface is built by extending the **Ryven** framework's frontend components. While Ryven provides a reference implementation, GLIDER integrates the ryvencore-qt library directly. This library provides the specialized QGraphicsItem implementations for Nodes and Connections.

To ensure usability, the editor supports:

- **Drag-and-Drop Library:** A side panel listing all available Hardware and Logic nodes.
- **Visual Debugging:** Active connections animate or change color to indicate data flow (e.g., a wire turns green when a signal is high).
- **Inline Widgets:** Nodes contain embedded Qt widgets (e.g., a slider inside a "PWM Control" node) using QGraphicsProxyWidget, allowing users to adjust parameters directly on the graph.[23]

---

4. The Flow Engine and Logic Core

The heart of GLIDER is the logic engine that executes the experiment. This engine must be robust, deterministic, and capable of handling complex dependencies.

4.1 Ryven Core Integration

GLIDER utilizes ryvencore, the backend library of Ryven, to manage the graph state.[13] Ryven's architecture is uniquely suited for this project because it natively supports the concept of distinct "Data" and "Execution" flows, a feature often missing in simpler dataflow libraries.

In ryvencore, a Node is a Python class. The graph execution is managed by event propagation.

- **Data Updates:** When a node calls self.set_output_val(index, value), the engine immediately pushes this value to connected inputs and triggers an update_event() on the downstream node. This recursive propagation handles the reactive logic (e.g., Sensor -> Math -> Plot).
- **Execution Signals:** When a node calls self.exec_output(index), it triggers a specific event on the connected node. This allows for conditional logic (e.g., IF Node: True -> Trigger A, False -> Trigger B).

4.2 Custom Node Architecture

GLIDER extends the base ryvencore.Node class to create a GliderNode. This subclass adds functionality specific to experimental control, such as error handling state, hardware device binding, and serialization helpers.

Key Node Categories:

1. **Hardware Nodes:** Proxies for physical devices (e.g., "Digital Pin Write", "DHT22 Read"). These nodes hold a reference to the specific hardware driver instance.
2. **Logic Nodes:** Mathematical operations, comparisons, timers, and PID controllers.
3. **Interface Nodes:** Dashboard widgets (e.g., "Gauge", "Chart", "Toggle Switch") that expose controls to the Runner UI.
4. **Script Nodes:** A special node type allowing the user to write arbitrary Python code within the GUI. This code is executed dynamically, offering maximum flexibility for complex, non-standard logic.[14]

4.3 Serialization and Experiment Files

Experiments are saved as JSON files. The choice of JSON over binary formats (like Pickle) ensures human readability and version control compatibility (Git).[24] GLIDER employs **JSON Schema** to validate these files, ensuring that an experiment file created on a desktop machine corresponds to a valid structure before being loaded onto a Raspberry Pi.[25]

The JSON structure separates the *Hardware Configuration* (Pin mappings, board types) from the *Flow Logic* (Nodes, connections). This allows the logic to be reused across different hardware setups by simply re-mapping the hardware configuration section.

Schema Structure:

- **metadata**: Version, Author, Date.
- **hardware**: List of Board definitions (Driver type, Port) and Device definitions (Name, Type, Pin config).
- **flow**: List of Nodes (Type, Position, State) and Connections (From_ID, To_ID).
- **dashboard**: Configuration of the Runner UI layout (which nodes are visible and where).

---

5. Hardware Abstraction Layer (HAL)

The HAL is the interface between the high-level Flow Engine and the low-level physical world. Its primary goal is to provide a uniform API for diverse hardware, enabling the software to treat a "Digital Output" on a Raspberry Pi exactly the same as one on an Arduino or a dedicated DAQ card.

5.1 Abstract Base Classes (ABCs)

To enforce consistency, GLIDER defines a set of Abstract Base Classes using Python's abc

module.[4]

The Board Interface:

Python

```
class BaseBoard(ABC):
    @abstractmethod
    async def connect(self):...
    @abstractmethod
    async def disconnect(self):...
    @abstractmethod
    async def set_pin_mode(self, pin, mode):...
    @abstractmethod
    async def write_digital(self, pin, value):...
    @abstractmethod
    async def read_analog(self, pin):...
```

Any hardware plugin must implement this interface. This polymorphism allows the Core to iterate over a list of BaseBoard objects and perform operations without knowing the specific hardware implementation details.

The Device Interface:
Devices represent higher-level components attached to the board (e.g., "Stepper Motor", "Temperature Sensor"). They wrap the BaseBoard methods into semantic actions (e.g., motor.step(), sensor.get_temp()).

5.2 Board Implementations

Arduino (Telemetrix-AIO):
For Arduino communication, GLIDER utilizes the Telemetrix-AIO library. Unlike the older pyfirmata, Telemetrix is actively maintained and supports asynchronous operation natively.27 It communicates with the Arduino over serial (USB) using a custom protocol that allows for callback-based reporting. This means the software doesn't need to constantly poll the Arduino; the Arduino pushes data when pins change, which is far more efficient for the event loop.
Raspberry Pi (GPIO):
For direct GPIO control on the Pi, GLIDER uses libraries like lgpio or gpiozero. Since these libraries often use blocking calls or their own threading models, the GLIDER HAL wraps these

calls using asyncio.to_thread() to ensure they do not block the main event loop.29 This wrapper layer harmonizes the blocking local calls with the non-blocking async calls used for remote boards.

5.3 Device Assignment and Pin Management

A common source of errors in experimental setups is pin conflict (e.g., assigning two devices to Pin 13). The GLIDER HAL includes a **Pin Manager** that tracks the allocation of resources. When a user attempts to assign a "Beam Break Sensor" to Pin 4, the Pin Manager checks if Pin 4 is already claimed by another device. If so, it raises an error in the GUI.

Furthermore, the Plugin system allows boards to define their capabilities via a "Capabilities Map" (e.g., "Pin A0 supports Analog Input, but not PWM"). The GUI uses this map to filter available pins in dropdown menus, preventing invalid configurations.[30]

---

6. Concurrency and Performance

Orchestrating hardware requires precise timing. If the UI thread freezes while rendering a complex graph, a motor might overrun its limit switch. GLIDER's concurrency model addresses this risk.

6.1 The Asyncio-Qt Bridge

The integration of asyncio with PyQt6 is achieved through **qasync**. This library provides a custom QEventLoop that acts as a drop-in replacement for the standard asyncio loop but pumps Qt events (mouse clicks, redraws) within the same cycle.[10]

Implementation Strategy:

1. **Startup:** The application creates a QApplication instance and a qasync.QEventLoop.
2. **Execution:** The loop.run_forever() method is called. This handles both UI responsiveness and async tasks.
3. **Tasks:** Hardware operations are scheduled as asyncio.Task objects. A "Flow Execution" is essentially a chain of awaited coroutines.

This architecture ensures that waiting for a 500ms delay in an experiment flow (await asyncio.sleep(0.5)) releases control back to the event loop, allowing the GUI to remain perfectly responsive during the wait.[31]

6.2 Managing Throughput and Backpressure

High-frequency sensors (e.g., reading an accelerometer at 100Hz) can flood the event loop if every data point triggers a UI update (which runs at screen refresh rate, typically 60Hz).

Decoupled UI Update Strategy:

GLIDER implements a Producer-Consumer pattern for data visualization.

- **Producer (Hardware):** The hardware driver reads data as fast as configured (e.g., 100Hz) and pushes it into a circular buffer (using collections.deque).
- **Consumer (UI):** A QTimer running at a human-perceptible rate (e.g., 20Hz) queries the buffer. It takes the latest value (or an average of the new values) and updates the label or plot.

This decoupling ensures that the overhead of drawing pixels does not throttle the data acquisition rate, ensuring high-performance logging alongside smooth visualization.[32]

---

7. Scalability and Modularity

Scalability in GLIDER refers to the ability to manage increasing complexity in both the size of the experiment (number of nodes) and the diversity of hardware.

7.1 Plugin Discovery Mechanism

The plugin system uses a directory-based discovery approach. The core application scans a plugins/ directory at startup. Each sub-directory is expected to contain a manifest.json describing the plugin (Name, Version, Entry Point) and the Python source.

This allows the community to share plugins easily. A user can download a keithley_multimeter folder, drop it into the plugins directory, and GLIDER will automatically register the new board and its associated device nodes. This is implemented using importlib.util.spec_from_file_location to dynamically load modules without hardcoding import paths.[34]

7.2 Dependency Management

Hardware drivers often require specific third-party libraries (pyserial, smbus2). To avoid "dependency hell," plugins define their requirements in a standard requirements.txt file. GLIDER includes a utility to parse these requirements and, upon user confirmation, install them into the active Python environment using pip calls from within the application (or creating a virtual environment dedicated to the application).

---

8. Deployment and Environment Configuration

8.1 Raspberry Pi Kiosk Deployment

To function effectively as an appliance on the Raspberry Pi, GLIDER must take over the user experience.

Autostart Configuration:
The recommended deployment uses the LXDE autostart mechanism or a Systemd User Service to launch GLIDER immediately after the X server starts.
Command: /usr/bin/python3 -m glider --mode runner --fullscreen
Screen Configuration:
The 480x800 resolution is non-standard for many desktop environments. The deployment documentation must specify the /boot/config.txt settings (e.g., display_rotate=1 or dtoverlay=vc4-kms-dsi-7inch) required to ensure the touchscreen is correctly oriented and calibrated.5

8.2 Packaging

For distribution, GLIDER utilizes **PyInstaller** to bundle the Python interpreter, Qt libraries, and the Core application into a single executable for Windows and Linux desktops. For the Raspberry Pi, distribution via a Python Wheel (.whl) is preferred. This allows GLIDER to leverage the system-optimized versions of libraries like numpy and RPi.GPIO provided by the Raspberry Pi OS package repository, avoiding binary incompatibility issues.[35]

---

9. Conclusion

The GLIDER design document presents a comprehensive blueprint for a next-generation experimental orchestration tool. By rejecting the monolithic script paradigm in favor of a modular, event-driven architecture, GLIDER solves the fundamental tension between ease of use and technical flexibility.

The selection of **PyQt6** provides the necessary graphical fidelity and cross-platform robustness. The integration of **Ryven** offers a proven, powerful logic engine that aligns with the mental model of experimental flow. The adoption of **asyncio** and **Telemetrix** ensures that the system meets the rigorous timing demands of hardware control without sacrificing interface responsiveness. Finally, the strict **Plugin Architecture** guarantees that GLIDER is not a static tool, but an evolving platform capable of growing with the needs of the scientific community. This architecture ensures that a researcher can design a complex experiment on a PC and deploy it confidently to a Raspberry Pi, bridging the gap between design and execution in the laboratory.

Detailed Technical Implementation Plan

1. Core System Architecture

1.1 Class Hierarchy and MVC Implementation

The structural integrity of GLIDER relies on a defined hierarchy that enforces the separation of the data model from the user interface.

Table 2: Core Class Definitions

Class	Type	Responsibility	Dependencies
GliderCore	Controller	The central orchestrator. Initializes the event loop, loads plugins, manages the ExperimentSession.	asyncio, importlib
ExperimentSession	Model	Represents the current state: the Hardware Map (Boards/Devices) and the Logic Graph. Serializable to JSON.	json, dataclasses
HardwareManager	Sub-Controller	Manages the lifecycle of connections. Maintains the registry of active Board instances.	telemetrix, gpiozero
FlowEngine	Sub-Controller	Wraps the ryvencore session. Handles the execution of the graph logic.	ryvencore
MainWindow	View	The primary PyQt6 window. Manages the high-level layout (Stack) and view switching.	PyQt6

NodeGraphView	View	Renders the visual flow editor. Handles zoom, pan, and drag interactions.	ryvencore-qt
RunnerDashboard	View	Renders the touch-optimized runtime interface for the Pi.	PyQt6

1.2 Event Loop Integration Strategy

The most complex technical challenge is unifying the blocking nature of the Qt exec() loop with the cooperative asyncio loop.

Implementation Detail:
The entry point of the application (main.py) must instantiate the QApplication first, then the qasync.QEventLoop.

Python

```
import sys
import asyncio
from PyQt6.QtWidgets import QApplication
from qasync import QEventLoop
from glider.core import GliderCore
from glider.gui import MainWindow

def main():
    # 1. Initialize Qt Application
    app = QApplication(sys.argv)

    # 2. Initialize QAsync Event Loop
    loop = QEventLoop(app)
    asyncio.set_event_loop(loop)

    # 3. Bootstrap Core and GUI
```

```
    core = GliderCore()
    window = MainWindow(core)
    window.show()

    # 4. Run the hybrid loop
    # This runs the Qt event dispatcher AND the asyncio scheduler
    with loop:
        loop.run_forever()
```

This pattern ensures that any await call within the application yields control back to the Qt event dispatcher, keeping the GUI fluid even during network requests or hardware waits.[10]

---

2. Hardware Abstraction Layer (HAL) Specifications

The HAL is designed to maximize code reuse across different hardware platforms.

2.1 The Plugin Interface (ABC)

Plugins must inherit from BaseBoard and implement all abstract methods. The BaseBoard class enforces the asynchronous contract.

BaseBoard Definition:

- name (str): Unique identifier (e.g., "Arduino Uno").
- async connect() -> bool: Establishes the physical link.
- async disconnect() -> None: Cleanly shuts down the link.
- async write_pin(pin: int, type: PinType, value: Any) -> None: Generic write method.
- async read_pin(pin: int, type: PinType) -> Any: Generic read method.
- pin_map (Dict): A property returning the capabilities of the board (e.g., identifying PWM-capable pins).

2.2 Telemetrix (Arduino) Implementation

The TelemetrixBoard class wraps telemetrix_aio.TelemetrixAIO.

- **Initialization:** It accepts a COM port (or auto-detects).
- **Callback Handling:** Telemetrix uses callbacks for data reporting. The TelemetrixBoard registers a universal internal callback that routes data to the appropriate Device object in GLIDER based on the pin number.
- **Async/Await:** Methods like board.set_pin_mode_digital_output are awaited directly, preserving the non-blocking architecture.[28]

2.3 GPIO (Raspberry Pi) Implementation

The PiGPIOBoard class wraps gpiozero or lgpio.

- **Thread Safety:** Since gpiozero callbacks run in a separate thread, the PiGPIOBoard must use loop.call_soon_threadsafe(callback, data) to marshal the data back into the main GLIDER event loop. This prevents the "QObject: Cannot create children for a parent that is in a different thread" error common in PyQt applications.[29]

2.4 Device Definition

Devices are logical entities that abstract pin numbers into functionality.

BaseDevice:

- config: Dictionary containing pin assignments (e.g., {'trigger_pin': 12, 'echo_pin': 13}).
- actions: Dictionary mapping command strings to methods (e.g., {'activate': self.turn_on}).
- This structure allows the Flow Engine to trigger device actions generically using the actions map without needing to know the specific class type of the device.

---

3. Flow Engine Implementation

3.1 Node Architecture

Nodes in GLIDER extend ryvencore.Node. They act as the bridge between the visual graph and the Python logic.

Visualizing Logic:
The ryvencore-qt library provides the NodeItem class which handles the painting of the node. GLIDER customizes this to match a dark, scientific aesthetic (Dark Gray backgrounds, distinct colored borders for different node categories: Green for Hardware, Blue for Logic, Orange for Interface).
Script Node Implementation:
The "Script Node" allows users to input custom Python code.

- **Storage:** The code is stored as a string property of the node.
- **Execution:** The node uses Python's exec() function.
- **Context:** The exec() context is populated with the node's inputs (inputs, etc.) and a reference to the glider API.
- **Security:** While exec() is dangerous, in a research context, power is prioritized over restriction. Warnings will be displayed when loading experiments containing script nodes.

3.2 Data Serialization (JSON Schema)

The save file acts as the single source of truth.

JSON Structure Example:

JSON

```
{
 "glider_version": "1.0.0",
 "hardware": {
  "boards":,
  "devices":
 },
 "flow": {
  "nodes":}
  ],
  "connections":
 }
}
```

The separation of hardware and flow allows the user to replace a board definition (e.g., swap Arduino for Pi) without breaking the flow logic, provided the new device instance maintains the same UUID.

---

4. User Interface Implementation

4.1 View Management

The application creates a QStackedWidget as the central widget of the QMainWindow.

- **Index 0 (Builder):** Contains the NodeGraphView (Graphics View).
- **Index 1 (Runner):** Contains the RunnerWidget (Scroll Area).

Switching Logic:
A toolbar button "Switch View" allows manual toggling. Additionally, a startup argument --mode runner forces the application to start in Index 1, useful for the Raspberry Pi autostart configuration.

4.2 The Runner Dashboard

The Runner Dashboard is constructed dynamically based on the nodes in the flow.

- Nodes marked as visible=True in the editor appear here.

- **Widgets:** GLIDER provides a set of RunnerWidgets—simplified Qt widgets styled for touch.
    - TouchButton: A QPushButton with a minimum height of 80px and CSS styling for a "pressed" state.
    - TouchLabel: A QLabel with large, high-contrast font (24pt).
    - TouchPlot: A simplified pyqtgraph widget for real-time data visualization, stripping away complex menus to save screen real estate.

4.3 Stylesheets (QSS)

Two distinct .qss files are maintained:

1. desktop.qss: Standard padding, system default fonts, scrollbar sizes optimized for mouse pointers.
2. touch.qss:
   CSS
   ```
   QScrollBar:vertical { width: 40px; }
   QPushButton { padding: 20px; font-size: 18pt; }
   QLabel { font-size: 16pt; }
   ```

   This ensures usability on the resistive or capacitive touchscreens common with Raspberry Pis.[20]

---

5. Deployment and Maintenance

5.1 Installation Strategy

GLIDER will use a pyproject.toml file to define dependencies.
For the Raspberry Pi, a shell script install_pi.sh will be provided:

1. Updates system packages (apt update).
2. Installs system-level dependencies (libqt6...).
3. Creates a Python virtual environment.
4. Installs GLIDER via pip.
5. Prompts to enable the systemd service for autostart.

5.2 Documentation

Documentation will be generated using **Sphinx**. It will include:

- **User Guide:** How to build flows.
- **Developer Guide:** How to write hardware plugins (ABCs).
- **API Reference:** Auto-generated from docstrings.

5.3 Error Handling and Recovery

Hardware in the loop implies instability (wires get pulled, power fails).

- **Reconnection Logic:** The BaseBoard includes a auto_reconnect flag. If a serial exception occurs, the Core attempts to reconnect every 2 seconds, notifying the GUI via a non-blocking signal.
- **Safe State:** On any unrecoverable error, the Core triggers a global emergency_stop() signal, which invokes the shutdown() method on all registered devices, ensuring motors stop and heaters turn off.[2]

6. Conclusion

The GLIDER architecture provides a comprehensive solution for modern experimental orchestration. By leveraging the specific strengths of **PyQt6** for visualization, **Ryven** for logic definition, and **asyncio/qasync** for concurrency, it successfully navigates the trade-offs between ease of use and technical capability. The result is a platform that empowers researchers to construct complex, hardware-integrated experiments with the same ease as drawing a flowchart, scalable from a laboratory workbench to a deployed embedded sensor station.

Works cited

1. Python and Embedded Systems: Running Python on Raspberry Pi, Arduino, or Microcontrollers | by Code With Hannan | Medium, accessed December 26, 2025, https://medium.com/@CodeWithHannan/python-and-embedded-systems-running-python-on-raspberry-pi-arduino-or-microcontrollers-3f8d33028eec
2. How to Install the GPIO Python Library - The Pi Hut, accessed December 26, 2025, https://thepihut.com/blogs/raspberry-pi-tutorials/how-to-install-the-gpio-python-library
3. Physical Computing with Python - Code Club Projects - Raspberry Pi Foundation, accessed December 26, 2025, https://projects.raspberrypi.org/en/projects/physical-computing
4. Python Abstract Classes: A Comprehensive Guide with Examples - DataCamp, accessed December 26, 2025, https://www.datacamp.com/tutorial/python-abstract-classes
5. touchpi is a Python project that simplifies GUI development for Raspberry Pi devices with touch displays. - GitHub, accessed December 26, 2025, https://github.com/touchpi/touchpi
6. Raspberry Pi 5 (Bookworm) Kiosk Mode with Python Script, accessed December 26, 2025, https://forums.raspberrypi.com/viewtopic.php?t=393187
7. Simplest way of deploying a Python application to a Raspberry Pi : r/raspberry_pi - Reddit, accessed December 26, 2025, https://www.reddit.com/r/raspberry_pi/comments/x7a1xp/simplest_way_of_deploy

ing_a_python_application_to/
8. Sync or Async? Exploring Single vs. Multi-Threading Execution Models in Depth - Medium, accessed December 26, 2025, https://medium.com/@karam.majdi33/sync-or-async-exploring-single-vs-multi-threading-execution-models-in-depth-775c1a481fe9
9. CabbageDevelopment/qasync: Python library for using asyncio in Qt-based applications., accessed December 26, 2025, https://github.com/CabbageDevelopment/qasync
10. qasync - PyPI, accessed December 26, 2025, https://pypi.org/project/qasync/
11. pyQt6 + asyncio: connect an async function as a slot - Stack Overflow, accessed December 26, 2025, https://stackoverflow.com/questions/74196406/pyqt6-asyncio-connect-an-async-function-as-a-slot
12. Creating and discovering plugins - Python Packaging User Guide, accessed December 26, 2025, https://packaging.python.org/guides/creating-and-discovering-plugins/
13. samuelwoelfl/Ryven-Website-2.0: New, fresh website for the flow-based visual programming solution "Ryven" by @leonthomm - GitHub, accessed December 26, 2025, https://github.com/samuelwoelfl/Ryven-Website-2.0
14. Ryven - Flow-based visual scripting for Python, accessed December 26, 2025, https://ryven.org/
15. Which GUI Framework is the best for Python coders? - ActiveState, accessed December 26, 2025, https://www.activestate.com/blog/top-10-python-gui-frameworks-compared/
16. 7 Tools for GUI Development on Raspberry Pi | by Sasha Shturma | Women Make | Medium, accessed December 26, 2025, https://medium.com/women-make/7-tools-for-gui-development-on-raspberry-pi-67fd7cd9226e
17. QtQuick or QGraphicsView - Stack Overflow, accessed December 26, 2025, https://stackoverflow.com/questions/11977766/qtquick-or-qgraphicsview
18. The Qt Graphics View Framework. - LearnQt, accessed December 26, 2025, https://www.learnqt.guide/qt-graphics-view-framework
19. QWidgets vs QML: performance and touch events management : r/QtFramework - Reddit, accessed December 26, 2025, https://www.reddit.com/r/QtFramework/comments/17vq9f0/qwidgets_vs_qml_performance_and_touch_events/
20. Styling PyQt6 Applications - Default and Custom QSS Stylesheets - Stack Abuse, accessed December 26, 2025, https://stackabuse.com/styling-pyqt6-applications-default-and-custom-qss-stylesheets/
21. Qt Style Sheets - Qt for Python, accessed December 26, 2025, https://doc.qt.io/qtforpython-6.5/overviews/stylesheet.html
22. Getting Started with PyQt6: A Beginner-Friendly Guide to Modern GUI Development, accessed December 26, 2025, https://medium.com/@jasonyang.algo/getting-started-with-pyqt6-a-beginner-fri

endly-guide-to-modern-gui-development-79924151440f
23. Ryven - A simple visual node editor for Python - CodeSandbox, accessed December 26, 2025, http://codesandbox.io/p/github/sanskarsakya/Ryven
24. What Is JSON Schema? | Postman Blog, accessed December 26, 2025, https://blog.postman.com/what-is-json-schema/
25. Creating your first schema - JSON Schema, accessed December 26, 2025, https://json-schema.org/learn/getting-started-step-by-step
26. abc — Abstract Base Classes — Python 3.14.2 documentation, accessed December 26, 2025, https://docs.python.org/3/library/abc.html
27. MrYsLab/telemetrix: A user-extensible replacement for StandardFirmata. All without the complexity of Firmata! - GitHub, accessed December 26, 2025, https://github.com/MrYsLab/telemetrix
28. Telemetrix and Telemetrix-AIO User's Guide, accessed December 26, 2025, https://mryslab.github.io/telemetrix/
29. Get to know Asynchio: Multithreaded Python using async/await - Daily.dev, accessed December 26, 2025, https://daily.dev/blog/get-to-know-asynchio-multithreaded-python-using-asyncawait
30. Python PyQt comboBox example - w3resource, accessed December 26, 2025, https://www.w3resource.com/python-exercises/pyqt/python-pyqt-connecting-signals-to-slots-exercise-5.php
31. A Conceptual Overview of asyncio — Python 3.14.2 documentation, accessed December 26, 2025, https://docs.python.org/3/howto/a-conceptual-overview-of-asyncio.html
32. QWidget::update() efficiency with hierarchy of custom widgets - Qt Centre, accessed December 26, 2025, https://www.qtcentre.org/threads/38182-QWidget-update()-efficiency-with-hierarchy-of-custom-widgets
33. Real-Time Logging with PyQt6. Introduction | by NugrohoW | Medium, accessed December 26, 2025, https://medium.com/@nugroho.wijatmiko/real-time-logging-with-pyqt6-78410e67860c
34. Plugin Architecture for Python - Binary Coders - WordPress.com, accessed December 26, 2025, https://binarycoders.wordpress.com/2023/07/22/plugin-architecture-for-python/
35. Installing PyQt6 on Raspberry Pi - python - Stack Overflow, accessed December 26, 2025, https://stackoverflow.com/questions/75509805/installing-pyqt6-on-raspberry-pi