**Laser Map Explorer (LaME) + Blockly — Handover Guide**

Developer Documentation

September 15, 2025

# Overview

This guide explains how the “Laser Map Explorer (LaME) in Blockly” project is put together, what each file does, and how everything runs end-to-end. It’s written for the next maintainer so you can extend the block set and keep shipping features with confidence.

# Big-picture architecture

Goal: Let users build LaME workflows visually in a Blockly canvas. The workspace generates Python that’s executed inside the LaME app context. Plots open in a Matplotlib-backed UI (either embedded or as a popup), and users can step through batch flows.

High-level flow:  
1) Blockly front-end (HTML + JS bundle) renders the toolbox, custom blocks, and generators.  
2) JavaScript generates Python from the current workspace and sends it to Python through Qt WebChannel.  
3) Workflow dock (PyQt6) receives the code via a bridge object and displays/executes it.  
4) LameBlockly executes the code inside LaME’s runtime (AppData, StyleData, plotting, clustering, etc.), then renders plots in a reusable CanvasDialog/CanvasWidget.

# Files & what they do

Front-end (Blockly) side:  
• index.html — Hosts the #blocklyDiv, toolbox categories, and loads qwebchannel.js + dist/bundle.js.  
• custom\_blocks.js — Defines all custom blocks (inputs, dropdowns, colors, tooltips, statement chains). Uses validators and dynamic dropdowns via window.blocklyBridge.  
• helper\_functions.js — Shared helpers for updating dropdowns, injecting default styling, managing the styling chain, histogram options, save path previews, and enable/disable reasons.  
• python\_generators.js — Blockly → Python codegen. Each block returns Python source that runs inside the LaME runtime (LameBlockly).

Python (PyQt6, LaME runtime) side:  
• Workflow.py — The Blockly dock & WebChannel bridge. Hosts the Blockly page, code preview, and exposes bridge slots.  
• BlocklyModules.py — LameBlockly runtime: executes generated Python, manages AppData/StyleData, clustering/DR, and figure display policy.  
• CanvasWidget.py — Matplotlib UI (tabs + toolbar) and a reusable popup dialog (CanvasDialog).  
• MainWindow.py — Application frame that installs the Workflow dock and global UI components.  
• LaME\_blockly\_design.docx — Block catalog & behavior template for remaining work.

# How it runs (end-to-end)

1) User builds a flow by dragging blocks. Blocks that depend on data start disabled; after a successful load\_sample/load\_directory, JS clears the disabled reason so the rest of the blocks can be used afterwards. (explained in detail in Block Enable/Disable Policy).  
2) The generator compiles the workspace to Python and calls runCode (preview) and/or executeCode (execute) via the WebChannel bridge.  
3) The Workflow dock passes the code to LameBlockly.execute\_code(...). Plots and analyses run inside the LaME runtime with access to models/utilities.  
4) Plotting uses ensure\_canvas\_popup() to reuse a shared CanvasDialog and CanvasWidget; figures are drawn there.  
5) Users interact with the canvas (pan/zoom/annotate, save figure/data) and continue or step through batch flows.

# Extending the system (adding a new block)

1) Design the block (name, category, inputs/outputs, tooltip, color). Add to the toolbox in index.html.  
2) Define it in custom\_blocks.js (validators, dynamic dropdowns, styling chain wiring).  
3) Implement a generator in python\_generators.js that sets app\_data/style\_data and calls the appropriate plotting/analysis methods.  
4) (Optional) Add WebChannel bridge slots in Workflow.py if the block needs Python-side data for dropdowns.  
5) Add runtime helpers in BlocklyModules.py if you’re calling new backend functionality (e.g., data extraction, plotting helpers).

# Operations to know

• Running the app — Launch the PyQt6 app. MainWindow adds the Workflow dock which loads the Blockly page.  
• Previewing generated code — The dock’s editor shows the Python; execute via the bridge.  
• Figure display policy — display\_figures=True shows a reusable CanvasDialog (Continue/Stop/Skip controls if implemented). Otherwise, embed figures into a layout.  
• Saving figures/data — The save\_plot block previews paths and triggers Python I/O; the canvas toolbar also has “Save Figure”.  
• Common pitfalls — Always use ensure\_canvas\_popup() before plotting to avoid C++ deletion errors. Disable blocks until data is loaded and re-enable them once ready.

# Where things live (quick map)

Front-end:  
• index.html — Toolbox and workspace host.  
• dist/bundle.js — Compiled JS (includes app.js, custom\_blocks.js, helper\_functions.js, python\_generators.js).

Python / Qt:  
• MainWindow.py — App frame.  
• Workflow.py — Dock UI + WebChannel registration and code preview/execution.  
• BlocklyModules.py — LameBlockly runtime.  
• CanvasWidget.py — Canvas widget + dialog.  
• LaME\_blockly\_design.docx — Source of truth for the block catalog.

# Continuing the block catalog

Use the design template to prioritize remaining blocks per category. Keep the color scheme and tooltips consistent. Ensure styling widgets flow to style\_data and are respected by plots.

# Appendix — Key API touchpoints

WebChannel (Workflow.py): register blocklyBridge, inject qwebchannel.js, load index.html, run/preview code, forward resizes.  
Bridge slots (Workflow.py): runCode(code) for preview; executeCode(code) to run inside LameBlockly.  
LameBlockly (BlocklyModules.py): App/Style state, plotting, clustering/DR, ensure\_canvas\_popup for robust figure display.  
Canvas toolkit (CanvasWidget.py): Reusable CanvasDialog, nav toolbar, multiple view tabs, and safe cleanup on close.

# Notes

This document summarizes the current architecture and conventions for running LaME inside a Blockly workspace. Use the LaME\_blockly\_design.docx template as the canonical reference for block behavior and coverage.

# Executing Blocks Individually

You can execute a single block directly by double‑clicking it in the workspace. When you double‑click a block, LaME runs that block and any blocks connected downstream (i.e., chained to its bottom connection). This is useful for quickly testing a portion of a workflow without running the entire script.

Notes:

• Double‑click executes the selected block first, then proceeds through the chain in order.

• If the block displays a figure, the configured display policy (popup vs. embed) applies.

• Any exceptions are reported with a traceback that includes the generated‑code line number.

# Block Enable/Disable Policy

For safety and clarity, most blocks are disabled by default until data is available. Only the File I/O blocks — “Load sample” and “Load directory” — are enabled at startup. All other blocks show a disabled reason: “Load sample to enable block.”

After you load a single sample or a directory of samples, the application enables the rest of the blocks so you can build and execute flows.

Implementation details (for maintainers):

• On startup, the UI iterates over workspace blocks and calls setDisabledReason('Load sample to enable block') for everything except File I/O blocks.

• After a sample is successfully loaded, the UI clears the disabled reason across the workspace (e.g., setDisabledReason(false)).

• If you add new blocks later, keep them disabled-by-default unless they are required to initiate loading.

# Debugging the Web UI (JS/HTML) via Chrome DevTools

The Blockly UI runs inside a PyQt6 QWebEngineView. You can attach Chrome DevTools to it using Qt WebEngine’s remote debugging. This allows you to inspect DOM, set breakpoints in JavaScript, and monitor console logs.

Steps:

1. Ensure the application sets the Qt WebEngine remote debugging port (e.g., 9222). Common approaches:

• Environment variable: QTWEBENGINE\_REMOTE\_DEBUGGING=9222

• Or pass the flag when initializing QWebEngine (depending on your app entry point).

2. Launch the app normally.

3. Open Chrome and navigate to: http://127.0.0.1:9222/

4. Click the entry that corresponds to the LaME/Blockly page to open DevTools.

5. In DevTools:

• Use the Elements/Console tabs to interact with the page.

• Set breakpoints under Sources (bundle.js and, if source maps are configured, the original modules).

• Inspect runtime state (e.g., window.blocklyBridge, workspace, and block instances).

Tip: If you use Webpack source maps in development builds, enable them to step through original files like custom\_blocks.js, helper\_functions.js, and python\_generators.js.