PRACTICAL ADVICE ON DEVELOPING RESEARCH SOFTWARE

PRACTICAL ADVICE ON DEVELOPING RESEARCH SOFTWARE BASED ON A CASE STUDY OF AN INTERACTIVE VISUALIZATION TOOL FOR BRAIN RESPONSE DATA

BY

BO LIANG, M.Eng.

a Report
submitted to the Department of Computing & Software
and the School of Graduate Studies
of McMaster University
in partial fulfilment of the requirements
for the degree of
Masters of Engineering

Masters of Engineering (2025)	McMaster University
(Department of Computing & Software)	Hamilton, Ontario, Canada

TITLE:	Practical Advice on Developing Research Software Based on a Case Study of an Interactive Visualization Tool for Brain Response Data
AUTHOR:	Bo Liang M.Eng. (Computing & Software Engineering), McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Christian Brodbeck, Dr. Spencer Smith
NUMBER OF PAGES:	xv, 79

Lay Abstract

Scientists today rely heavily on computer software to analyze data and visualize results. However, building high-quality research software that is maintainable and user-friendly remains challenging. This report examines how to develop better research software through a real-world case study: the creation of "LiveNeuron", an interactive tool that helps neuroscientists visualize brain activity data. By documenting the development process, this work explores practical strategies including following established design principles, conducting thorough code reviews with colleagues, gathering feedback from actual users, and creating clear guidelines for contributors. The goal is to bridge the gap between theoretical best practices and their real-world application. This report provides actionable insights for researchers and developers who create scientific software tools, demonstrating that good software design emerges from combining sound technical principles with collaborative teamwork and user input.

Abstract

This report presents a case study on the development of research software, detailing practical advice derived from the creation of "LiveNeuron," an interactive visualization tool for magnetoencephalography (MEG) and electroencephalography (EEG) data. The development of "LiveNeuron" as a modular package for the Eelbrain ecosystem serves as a lens through which to examine the real-world application of software engineering principles.

Key challenges addressed include architectural design, collaborative development practices, and user-centered evaluation. The report demonstrates how prioritizing modularity and isolating dependencies were crucial for managing complexity and ensuring long-term maintainability. It explores the role of code review, not just for quality assurance, but as a vital tool for collaborative design refinement and knowledge sharing, distinguishing between asynchronous reviews for routine changes and synchronous walkthroughs for complex problem-solving. Furthermore, the report underscores the importance of pragmatic application of design principles like DRY (Don't Repeat Yourself) and YAGNI (You Ain't Gonna Need It), treating them as guidelines rather than rigid rules.

The findings emphasize that the creation of high-quality, sustainable research software is a holistic process that integrates thoughtful design, robust collaborative workflows, clear contributor guidelines, and a strong focus on usability. The guidelines summarized in this report, grounded in the practical experience of developing "LiveNeuron," offer valuable insights for developers and research teams aiming to build impactful and maintainable scientific software.

Your Dedication

Optional second line

Acknowledgements

Acknowledgements go here.

Contents

List of Figures

List of Tables

Notation, Definitions, and Abbreviations

Notation

$\mathbf{y}$	Data variable representing brain activity measurements (NDVar)
t	Time variable
$\mathbb{R}^n$	n-dimensional real-valued space

Definitions

LiveNeuron	An interactive visualization tool developed as part of this project for exploring brain response data within the Eelbrain ecosystem
Research Software	Software developed to support scientific research activities, including data analysis, simulation, and visualization
Source Space	The set of locations in the brain where neural activity is estimated

from MEG/EEG measurements

Abbreviations

AI	Artificial intelligence
DRY	Don't Repeat Yourself
EEG	Electroencephalography
LSP	Liskov Substitution Principle
MEG	Magnetoencephalography
NDVar	N-Dimensional Variable (Eelbrain data structure)
SOLID	Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion
TRF	Temporal Response Function
UI	User Interface
YAGNI	You Aren't Gonna Need It

Declaration of Academic Achievement

The student will declare his/her research contribution and, as appropriate, those of colleagues or other contributors to the contents of the thesis.

Chapter 1

Introduction

This chapter introduces the project context and the case study that grounds the report. The work centers on the development of "LiveNeuron", an interactive visualization tool for brain response data that complements the Eelbrain ecosystem Brodbeck (2014).

To guide the reader, Section 1.1 presents the background and motivation for this study. Section 1.2 introduces the Eelbrain project and the "LiveNeuron" tool. Section 1.3 summarizes the objectives and contributions of this report and explains how the remaining chapters are organized.

1.1 Background and Motivation

Software is an indispensable tool in modern scientific research, facilitating data analysis, simulation, and visualization Wilson et al. (2014); Jiménez et al. (2017). As research questions evolve and new experimental techniques emerge, the software that supports these activities must also adapt.

The current state of scientific software development presents both opportunities and challenges. While scientific software has become increasingly sophisticated and integral to research outcomes Smith et al. (2016), its development often follows different practices than traditional software engineering. Scientific programmers, typically trained primarily as domain experts rather than software engineers, face unique challenges in balancing scientific innovation with software quality Hannay et al. (2009). This gap between domain expertise and software engineering knowledge can lead to sustainability and maintainability issues as projects grow in complexity Storer (2017).

However, extending existing scientific software with new functionalities is a non-trivial task Storer (2017); Segal (2005). It often exposes underlying challenges in the software's architecture and design. The process of adding a new feature can reveal issues related to modularity, scalability, and maintainability that might not have been apparent during the initial development Carver et al. (2007). Research has shown that scientific software development can benefit significantly from adopting software engineering practices such as systematic testing Kanewala and Bieman (2014), code review Petre and Wilson (2014), and modular design Wilson et al. (2017). This report is motivated by the conviction that a careful study of these challenges provides a valuable learning opportunity. By documenting and analyzing the design problems encountered in a real-world project, we can derive practical insights and principles that inform more robust and sustainable software development practices in the scientific domain Basili et al. (2008); Segal and Morris (2008).

1.2 Eelbrain Project Introduction

Eelbrain Brodbeck (2014) is an open-source Python library tailored for the analysis of time-continuous neurophysiological data, such as Magnetoencephalography (MEG) and Electroencephalography (EEG). It provides a powerful framework for statistical analysis based on Temporal Response Functions (TRFs), which allows researchers to model the brain's response to continuous stimuli like speech Brodbeck et al. (2018). By enabling the deconvolution of complex neural signals into distinct processing stages, Eelbrain has become a valuable tool in cognitive neuroscience.

The project described in this report sought to enhance Eelbrain's ecosystem by developing a new, interactive data visualization tool. The architectural decision was made to build this tool as a separate, self-contained package named "LiveNeuron". The development of "LiveNeuron" thus serves as a practical case study in designing modular software for a scientific computing ecosystem, focusing on challenges such as defining a clean interface and managing the internal state of the new package.

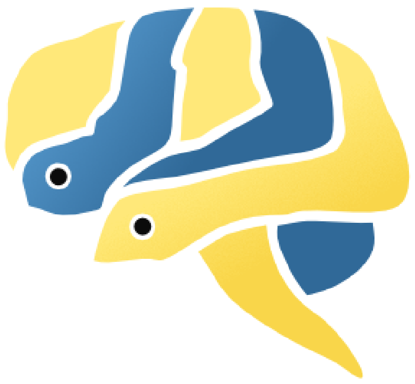

Figure 1.1: Eelbrain.

1.3 Objectives and Contributions of this Report

The central objective of this report is to provide a holistic analysis of the software development process, using the creation of the "LiveNeuron" package as a case study. It moves beyond a purely technical description to explore the critical roles of design principles, collaborative practices, and user-centered evaluation. This report will use the implementation of "LiveNeuron" as a case study to systematically trial a range of techniques, tools, and methods for developing research software, and to critically evaluate and reflect on their effectiveness and limitations in practice.

Specifically, we focus on four methods:

1. Explore the application of software design principles (such as SOLID, DRY, and YAGNI) to guide architectural decisions and maintain code quality in research software development.

2. Examine the function of code review as a mechanism for refining design, improving code quality, and reinforcing best practices within a development team Petre and Wilson (2014).

3. Investigate the impact of usability studies and direct user feedback on shaping a more effective and intuitive final product.

4. Discuss the importance of clear contributor guidelines in fostering sustainable and collaborative open-source scientific software Jiménez et al. (2017).

These four methods were selected because they represent complementary approaches that address different stages and aspects of the software development lifecycle. Design principles provide the foundational architectural guidelines for building maintainable code. Code review offers a collaborative quality assurance mechanism that catches issues early and promotes knowledge sharing among developers. Usability studies ensure that the resulting software meets the actual needs of end users, preventing the creation of technically sound but impractical tools. Finally, contributor guidelines establish a sustainable framework for long-term maintenance and community involvement, which is essential for open-source scientific software. Together, these methods form a comprehensive approach to developing high-quality research software that is well-designed, thoroughly vetted, user-friendly, and maintainable.

The main contribution of this report is to provide a reflective, experience-based account that integrates these different facets of software development. By grounding the discussion in a concrete project, it aims to bridge the gap between abstract principles and their real-world application, emphasizing that robust design is continually reinforced through collaborative practices like code review, offering valuable insights for those engaged in creating and maintaining scientific software tools.

The remainder of this report is organized as follows. Chapter 2 presents the design and implementation of the LiveNeuron package, detailing the architectural decisions and technical approach. Chapter 3 explores the application of software design principles, including SOLID, DRY, and YAGNI, demonstrating how these concepts guided development decisions. Chapter 4 examines the code review process, illustrating how collaborative review improved code quality and reinforced best practices. Chapter 5 discusses the usability study and the impact of user feedback on refining the tool's design and functionality. Finally, Chapter 6 synthesizes the key findings, reflects on lessons learned, and offers recommendations for future research software development efforts.

Chapter 2

Design and Implementation of LiveNeuron

This chapter presents the design and implementation of the LiveNeuron package, specifically focusing on the `EelbrainPlotly2DViz` class that provides interactive 2D brain visualization capabilities for the Eelbrain ecosystem. The discussion is organized to first establish the architectural context, then detail the core design decisions, and finally describe the implementation strategy with concrete examples from the codebase.

Section 2.1 provides an overview of how LiveNeuron fits within the broader Eelbrain architecture. Section 2.2 examines the key design decisions that shaped the package structure. Section 2.3 describes the implementation approach and the technologies employed. Section 2.4 discusses the technical challenges encountered and their solutions.

To provide an overview of the system's complexity and component interactions, Figure 2.1 illustrates the overall interaction logic flow in the LiveNeuron visualization system.

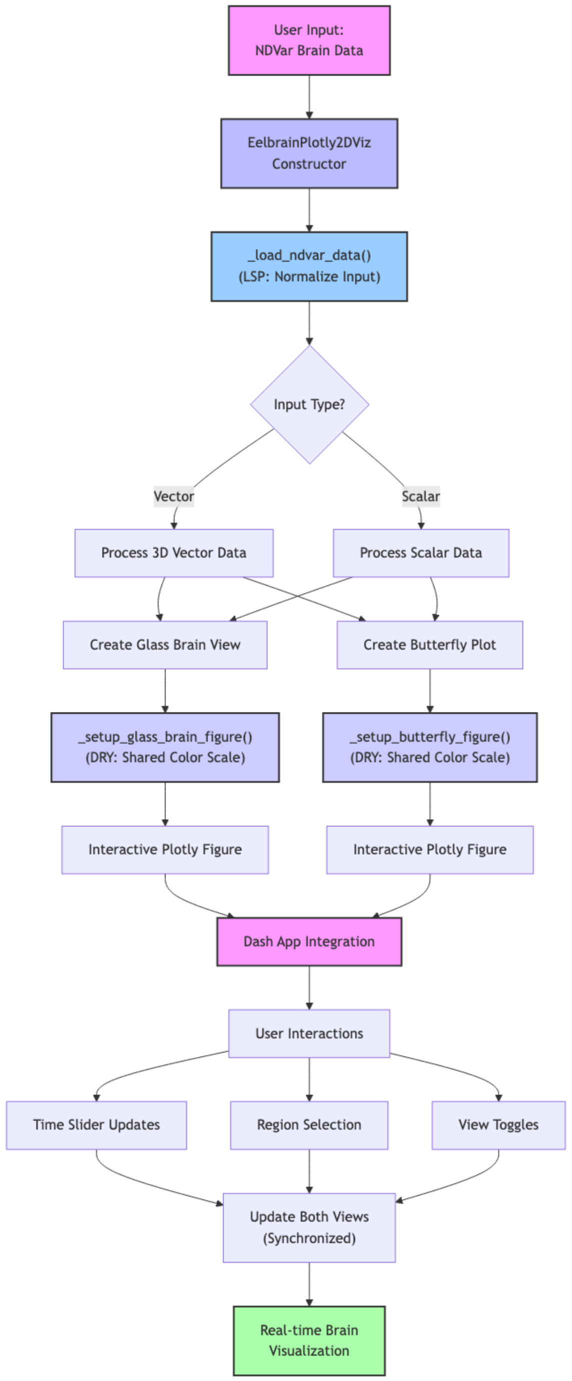

Figure 2.1: Overall Interaction Logic Flow in LiveNeuron System

2.1 Architecture Overview

LiveNeuron was designed as a standalone package that extends Eelbrain's capabilities rather than being integrated directly into the core library. This architectural decision reflects a modular design philosophy: by keeping LiveNeuron separate, we maintain clear boundaries of responsibility and allow independent evolution of both packages.

The interaction model between Eelbrain and LiveNeuron follows a layered approach. Eelbrain handles all data processing and analysis tasks, producing `NDVar` data structures that capture brain response information with dimensions such as `(case, time, source, space)`. LiveNeuron then consumes these data structures and provides interactive visualization capabilities through a web-based interface built with Dash and Plotly.

The primary class, `EelbrainPlotly2DViz`, accepts data in the same format as Eelbrain's existing `plot.GlassBrain` function, ensuring consistency with the broader ecosystem and minimizing the learning curve for existing users.

2.2 Design Decisions

Several critical design decisions shaped the development of LiveNeuron, each addressing specific challenges in scientific data visualization.

2.2.1 Package Modularity and Data Interface

The decision to develop LiveNeuron as a separate package required careful interface design. The `_load_ndvar_data()` method demonstrates this modularity by accepting Eelbrain's `NDVar` objects directly:

```
def _load_ndvar_data(self, y: NDVar) -> None:
    if y.has_case:
        y = y.mean("case")

    source = y.get_dim("source")
    self.source_coords = source.coordinates
    self.time_values = y.time.times
```

This design allows LiveNeuron to extract necessary information from Eelbrain's data structures without requiring deep integration into Eelbrain's internals. The interface supports both vector data (with a `space` dimension) and scalar data, adapting automatically to the input format.

2.2.2 Web-Based Visualization Framework

LiveNeuron employs Plotly Dash as its web-based visualization framework, chosen for several key advantages:

- **Cross-platform compatibility**: Works in standard web browsers without requiring platform-specific installations
- **Rich interactivity**: Supports hover events, click events, and dynamic updates
- **Jupyter integration**: Provides seamless embedding within Jupyter notebooks, a common environment for researchers
- **Pure Python development**: Eliminates the need for separate JavaScript development

The framework choice is reflected in the initialization:

```
self.app: dash.Dash = dash.Dash(__name__)
```

2.2.3 Multi-View Visualization Strategy

The visualization strategy employs multiple coordinated views to provide comprehensive data exploration:

1. **Butterfly Plot**: Displays time series data for all sources, showing temporal evolution of brain activity. The plot includes individual source traces, mean activity, and maximum activity across sources.

2. **2D Brain Projections**: Three orthogonal views (axial, sagittal, coronal) provide spatial context for the brain activity at a selected time point. These views use heatmaps for activity magnitude and arrow glyphs for vector directions.

The multi-view approach enables researchers to explore both temporal and spatial dimensions of their data within a single interface. Figure 2.2 shows the complete LiveNeuron visualization interface, demonstrating how the butterfly plot at the top is coordinated with the three 2D brain projection views below.

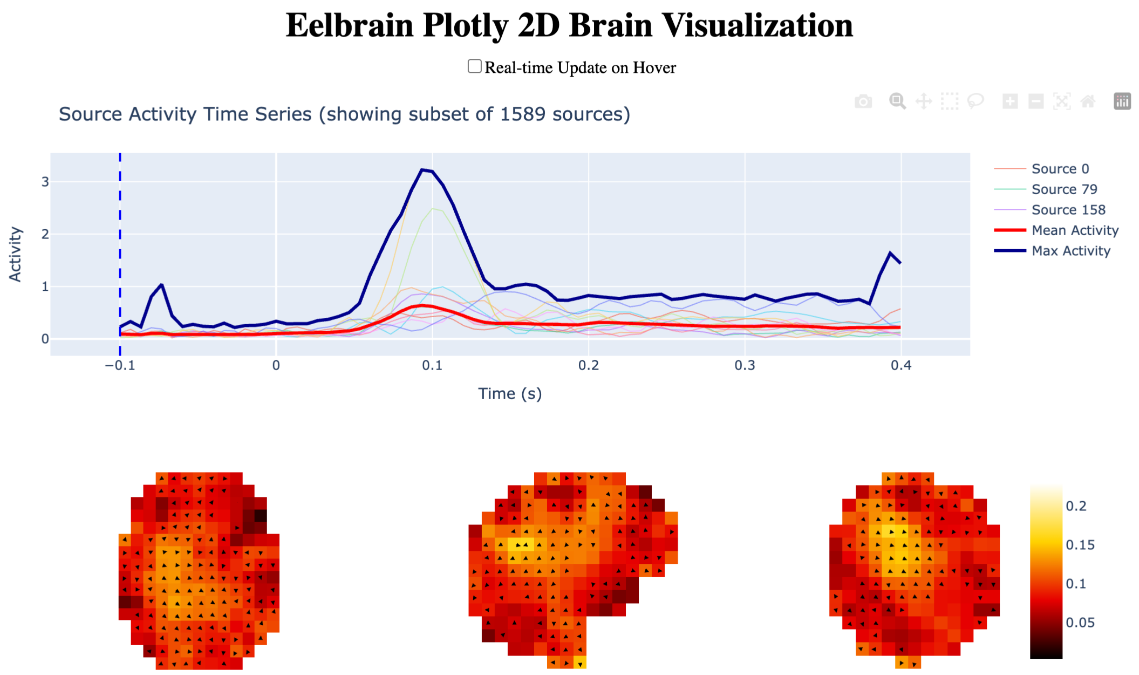
Eelbrain Plotly 2D Brain Visualization
Real-time Update on Hover
Source Activity Time Series (showing subset of 1589 sources)
Activity
Time (s)
Source 0
Source 79
Source 158
Mean Activity
Max Activity
0.2
0.15
0.1
0.05

Figure 2.2: LiveNeuron

2.2.4 State Management Architecture

Managing state in an interactive visualization presents unique challenges. LiveNeuron employs Dash's `dcc.Store` components for centralized state management:

```
dcc.Store(id="selected-time-idx", data=0),
dcc.Store(id="selected-source-idx", data=None),
```

This pattern provides several benefits:

- A single source of truth for application state
- Automatic synchronization between UI components
- Support for complex interaction patterns (hover vs. click)

- Separation of state from presentation logic

The callback system then orchestrates updates across different visualization components based on state changes.

2.2.5 Interaction Model

LiveNeuron implements a sophisticated interaction model with two distinct modes:

Click Mode (Default): Users click on the butterfly plot to select a specific time point, which updates the brain projection views. This mode provides stable, focused inspection of the data at chosen time points.

Real-time Hover Mode: When enabled, brain views update dynamically as users hover over the butterfly plot, providing immediate visual feedback for rapid data exploration. The implementation includes a safeguard where clicking in real-time mode automatically switches back to click mode for detailed inspection:

```
if "hoverData" in triggered_id and is_realtime:
    time_idx = np.argmin(np.abs(self.time_values - hover_time))
    return time_idx, dash.no_update, dash.no_update

if "clickData" in triggered_id:
    new_realtime_value = [] if is_realtime else dash.no_update
    return time_idx, source_idx, new_realtime_value
```

2.3 Implementation Details

2.3.1 Technology Stack

The implementation leverages several key technologies:

- **Plotly Dash**: Web application framework for building interactive visualizations
- **Plotly Graph Objects**: High-level plotting interface for creating interactive charts
- **NumPy**: Efficient array operations and numerical computing
- **Eelbrain NDVar**: Native data structure for neurophysiological data
- **Matplotlib**: Fallback rendering for image export functionality

2.3.2 Data Processing Pipeline

The data processing pipeline transforms Eelbrain's `NDVar` format into visualization-ready arrays:

1. **Case Averaging**: If data includes multiple cases, they are averaged: `y.mean("case")`
2. **Dimension Extraction**: Source coordinates and time values are extracted from the data dimensions
3. **Vector Handling**: For vector data with a `space` dimension, the implementation computes both the full 3D vectors and their magnitudes:

```
self.glass_brain_data = y.get_data(("source", "space", "time"))
self.butterfly_data = np.linalg.norm(self.glass_brain_data, axis=1)
```

4. **Scalar Adaptation**: For scalar data, the pipeline adapts by expanding dimensions to maintain a consistent internal representation

2.3.3 Butterfly Plot Implementation

The butterfly plot implementation addresses several practical challenges:

Performance Optimization: To maintain responsiveness with large datasets, the implementation limits the number of individual traces displayed while always showing mean and maximum activity:

```
if not self.show_max_only:
    max_traces = 20
    step = max(1, n_sources // max_traces)
    indices_to_plot = list(range(0, n_sources, step))
```

Auto-scaling: The plot automatically scales data to appropriate units (pA, nA, µA) based on magnitude, ensuring readability across different data ranges.

Clickable Background: A crucial design element is the invisible marker grid that enables clicking anywhere on the plot:

```
n_rows = 5
y_positions = np.linspace(y_min - y_margin/2, y_max + y_margin/2, n_rows)
x_grid = np.tile(self.time_values, n_rows)
y_grid = np.repeat(y_positions, len(self.time_values))
```

This creates a dense grid of transparent markers that capture click events throughout the plot area, greatly improving usability.

2.3.4 2D Brain Projection Rendering

The brain projection views employ several sophisticated techniques:

Adaptive Gridding: Rather than using a fixed grid, the implementation creates a data-driven grid that adapts to the actual source locations:

```
unique_x = np.unique(x_coords)
unique_y = np.unique(y_coords)
x_spacing = np.diff(unique_x).min() / 2 if len(unique_x) > 1 else 0.001
```

This ensures that each source occupies its own grid cell, preventing unwanted aggregation of nearby sources.

Consistent Color Scaling: The implementation computes global minimum and maximum values across all views to ensure consistent color mapping:

```
global_min = np.min(activity_magnitude)
global_max = np.max(activity_magnitude)
```

Weighted Averaging: When multiple sources fall into the same grid cell, the implementation computes the average activity rather than summing, preventing overemphasis of densely populated regions.

2.3.5 Vector Arrow Visualization

For vector data, arrows indicate direction and magnitude. The implementation includes several optimizations:

Threshold Filtering: Users can control arrow density through thresholding:

```
if self.arrow_threshold == "auto":
    threshold_value = 0.1 * np.max(arrow_magnitudes)
    show_arrow_mask = arrow_magnitudes > threshold_value
```

Position-based Deduplication: When multiple vectors exist at the same position, only the one with maximum magnitude is displayed:

```
if (pos_key not in position_to_max_idx or
    arrow_magnitudes[i] > arrow_magnitudes[position_to_max_idx[pos_key]]):
    position_to_max_idx[pos_key] = i
```

Batch Rendering: Instead of creating individual annotations for each arrow, the implementation uses batch traces that combine multiple arrows into single Plotly objects, dramatically improving rendering performance:

```
def _create_batch_arrows(self, fig, x_coords, y_coords,
                        u_vectors, v_vectors, arrow_scale):
    x_lines = []
    y_lines = []
    for i in range(len(x_coords)):
        x_lines.extend([x_coords[i], x_ends[i], None])
        y_lines.extend([y_coords[i], y_ends[i], None])

    fig.add_trace(go.Scatter(x=x_lines, y=y_lines, mode="lines"))
```

This optimization reduced rendering time from several seconds to near-instantaneous updates.

2.3.6 Jupyter Environment Detection and Adaptation

The implementation detects whether it's running in a Jupyter environment and adapts its layout accordingly:

```
def _is_jupyter_environment():
    try:
        from IPython import get_ipython
        return get_ipython() is not None
    except ImportError:
        return False
```

When running in Jupyter, the layout uses reduced heights and margins to better fit within notebook cells:

```
if self.is_jupyter_mode:
    butterfly_graph_style = {"height": "300px"}
    brain_height = "250px"
    margin = dict(l=30, r=30, t=30, b=30)
else:
    butterfly_graph_style = {"height": "400px"}
    brain_height = "450px"
    margin = dict(l=40, r=40, t=40, b=40)
```

2.4 Technical Challenges and Solutions

2.4.1 Performance with Large Datasets

Challenge: Rendering thousands of source points and vectors in real-time can cause significant performance degradation.

Solution: Multiple optimizations were implemented:

- Limiting the number of traces in the butterfly plot
- Batch rendering of arrow glyphs
- Using vectorized NumPy operations throughout
- Efficient state management to minimize unnecessary updates

2.4.2 Responsive Interaction Design

Challenge: Users need both quick exploration (hover) and detailed inspection (click), but combining these modes can lead to confusion.

Solution: The dual-mode interaction design with automatic mode switching provides both capabilities while maintaining intuitive behavior.

2.4.3 Cross-Environment Compatibility

Challenge: The tool needs to work seamlessly in both standard Python scripts and Jupyter notebooks, which have different display constraints.

Solution: Environment detection and adaptive layout generation ensure appropriate sizing and styling for each context, with a dedicated `show_in_jupyter()` convenience method for notebook users.

2.4.4 Data Heterogeneity

Challenge: Input data varies in format (vector vs. scalar), dimensionality (with or without case dimension), and scale.

Solution: The data loading pipeline automatically detects and adapts to different input formats, providing a consistent internal representation while preserving the flexibility to handle diverse data types.

Chapter 3

Design Principles in Research Software Development

Research software development presents unique challenges that distinguish it from traditional commercial software engineering. While commercial software often operates with fixed requirements and predictable use cases, research software must accommodate evolving scientific questions, experimental data formats, and computational methods. Yet despite these differences, the fundamental principles of good software design—modularity, maintainability, and clarity—remain essential for creating tools that serve the scientific community effectively.

This chapter explores how established software design principles apply to the domain of research software development, using concrete examples from the LiveNeuron visualization project. Section 3.1 examines the core objectives that drive software design decisions, establishing the foundation for understanding why design principles matter. Section 3.2 reviews three key design principles—SOLID, DRY, and

YAGNI—demonstrating their application through real code examples from the visualization system. Section 3.3 discusses the benefits and potential pitfalls of adhering to design principles, acknowledging the balance between theoretical ideals and practical constraints. Section 3.4 presents a detailed case study of how the LiveNeuron package evolved from an integrated feature to a standalone modular architecture, illustrating the real-world impact of design decisions on project success. Finally, Section 3.5 synthesizes lessons learned and offers practical recommendations for research software developers navigating similar challenges.

3.1 Core Goals of Software Design

At its core, software design is the process of envisioning and structuring a software solution to meet a set of requirements. While the immediate goal is always to create software that functions correctly, effective design looks beyond immediate functionality to achieve several long-term objectives. The most critical of these are maintainability, scalability, and usability.

Maintainability refers to the ease with which a software system can be modified to correct faults, improve performance, or adapt to a changed environment (Wilson et al., 2014). In scientific computing, where models and assumptions are constantly refined and new data formats emerge, maintainability is paramount. A maintainable system is well-organized, readable, and easy for new developers to understand, which reduces the cost and effort of long-term development.

Scalability is the measure of a system's ability to handle a growing amount of work by adding resources to the system (Basili et al., 2008). For scientific software, this often relates to performance with large datasets or complex computations. A scalable design does not just work for a small test case; it performs efficiently as the size of the input data or the number of users grows, as demonstrated in the case study in Chapter 3.

Usability concerns the ease with which a user can interact with the software to achieve their goals (Segal, 2005). In a scientific context, the users are often researchers who are experts in their domain but not necessarily in software engineering. A usable tool has an intuitive interface, provides clear feedback, and helps users avoid errors, ultimately making the research process more efficient and less frustrating.

3.2 Review of Common Design Principles

Several well-known software design principles provide a foundation for creating high-quality, maintainable software. As illustrated in the overall interaction logic flow (Figure 2.1 in Chapter 2), the LiveNeuron system demonstrates the application of these principles in practice.

These design principles include:

SOLID This is an acronym representing five core principles of object-oriented design (Martin, 2000):

- **Single Responsibility Principle**: A class should have only one reason to change, meaning it should have only one job or responsibility.
- **Open/Closed Principle**: Software entities should be open for extension but closed for modification.
- **Liskov Substitution Principle**: Objects of a superclass should be replaceable with objects of a subclass without breaking the application.

- **Interface Segregation Principle**: Clients should not be forced to depend on interfaces they do not use.
- **Dependency Inversion Principle**: High-level modules should not depend on low-level modules; both should depend on abstractions.

Together, they guide developers in creating systems that are easy to maintain and extend over time by promoting modularity and decoupling.

In the LiveNeuron visualization system, these principles are applied in several ways:

Single Responsibility Principle (SRP): Each method has a single, well-defined purpose. For example, the visualization class separates concerns into distinct methods:

```
class EelbrainPlotly2DViz:
    def _load_ndvar_data(self, y):
        # Responsibility: Load and normalize data
        ...

    def _setup_layout(self):
        # Responsibility: Configure UI layout
        ...

    def _setup_callbacks(self):
        # Responsibility: Wire interactive behaviors
        ...
    def create_butterfly_plot(self, time_idx):
        # Responsibility: Generate butterfly visualization
        ...

    def create_2d_brain_projections_plotly(self, time_idx):
        # Responsibility: Generate brain projection views
        ...
```

This separation ensures that each method can be tested, modified, and understood independently, reducing coupling and improving maintainability.

Open/Closed Principle: The class accepts configuration parameters in its constructor, allowing behavior customization without code modification:

```
def __init__(self, y=None, cmap="Hot", show_max_only=False,
             arrow_threshold=None):
    # Configuration injected at instantiation
    self.cmap = cmap  # Customize colormap
    self.show_max_only = show_max_only  # Control display mode
    self.arrow_threshold = arrow_threshold  # Filter arrows
```

Users can extend functionality by passing different parameters rather than modifying the class internals.

DRY (Don't Repeat Yourself) This principle is aimed at reducing the repetition of information or logic (Hunt and Thomas, 1999). By ensuring that every piece of knowledge has a single, unambiguous representation within a system, DRY makes the codebase more maintainable. When a change is needed, it only has to be made in one place, reducing the risk of inconsistencies. For example, in the LiveNeuron visualization, all brain views share one colorscale and one colorbar, serving as a single source of truth:

```
def render_views(views, cmap):
    # DRY: compute vmin/vmax once and reuse across all views.
    vmin = min(v.min() for v in views)
    vmax = max(v.max() for v in views)
    for v in views:
        draw_heatmap(v, cmap=cmap, vmin=vmin, vmax=vmax)
    # DRY: a single colorbar for all views
    draw_colorbar(vmin, vmax)
```

The DRY principle is demonstrated by computing color scale limits once and reusing them across all brain views, along with a single shared colorbar. Figure 3.1 illustrates this shared color scale logic flow, showing how a single computation of `vmin` and `vmax` propagates to all visualization components.

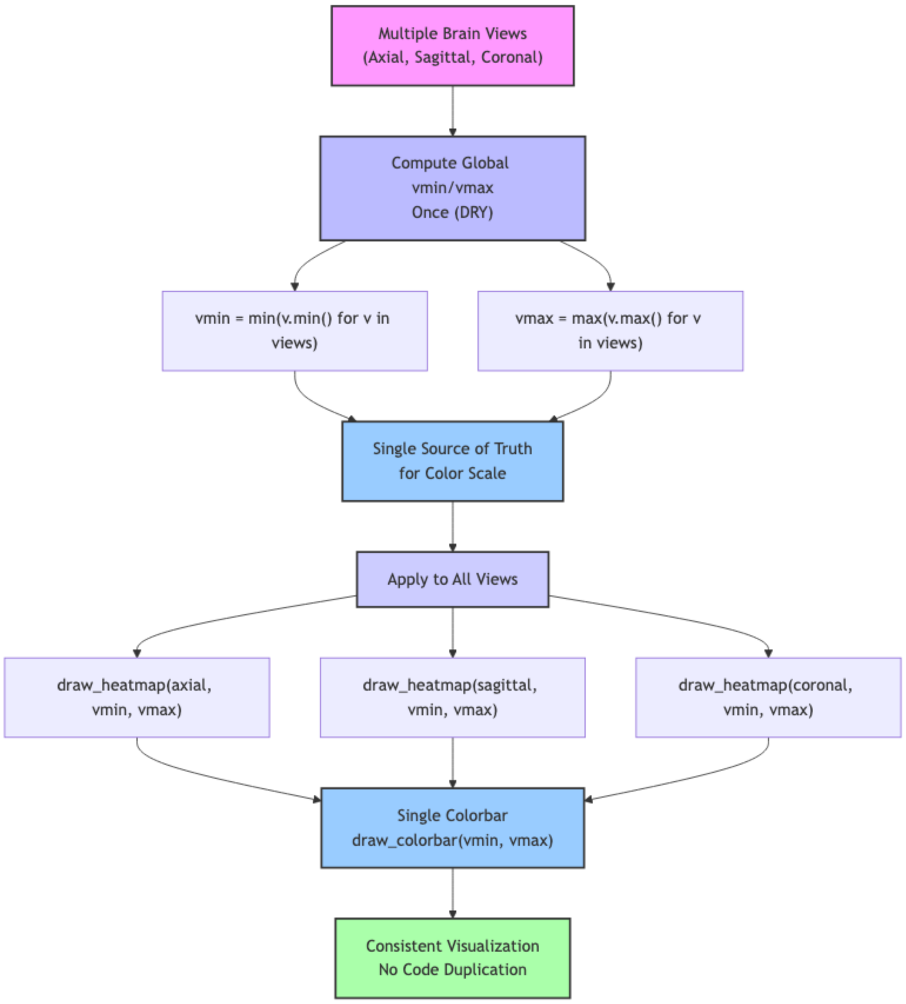
Multiple Brain Views
(Axial, Sagittal, Coronal)
Compute Global vmin/vmax Once (DRY)
vmin = min(v.min() for v in views)
vmax = max(v.max() for v in views)
Single Source of Truth for Color Scale
Apply to All Views
draw_heatmap(axial, vmin, vmax)
draw_heatmap(sagittal, vmin, vmax)
draw_heatmap(coronal, vmin, vmax)
Single Colorbar draw_colorbar(vmin, vmax)
Consistent Visualization No Code Duplication

Figure 3.1: DRY Principle: Shared Color Scale Logic in LiveNeuron

YAGNI (You Aren't Gonna Need It) A principle from agile development, YAGNI advises developers to implement only the functionality that is necessary for the immediate requirements. It cautions against adding features based on future speculation, as this can lead to code bloat and wasted effort on functionality that is never used. Example: omit a disk-loading parameter and accept only ready data:

```
# YAGNI: remove speculative parameter (data_source_location).
# Keep the API minimal to avoid unused complexity.
# avoid: def plot_brain(data, data_source_location=None, ...)
def plot_brain(data, *, region=None, cmap='Hot', show_max_only=False):
    ...  # no data_source_location
```

The YAGNI principle guides API design by avoiding speculative features, keeping the interface minimal until concrete requirements emerge. Figure 3.2 demonstrates this minimal API design logic, contrasting the avoided speculative approach with the clean, focused implementation.

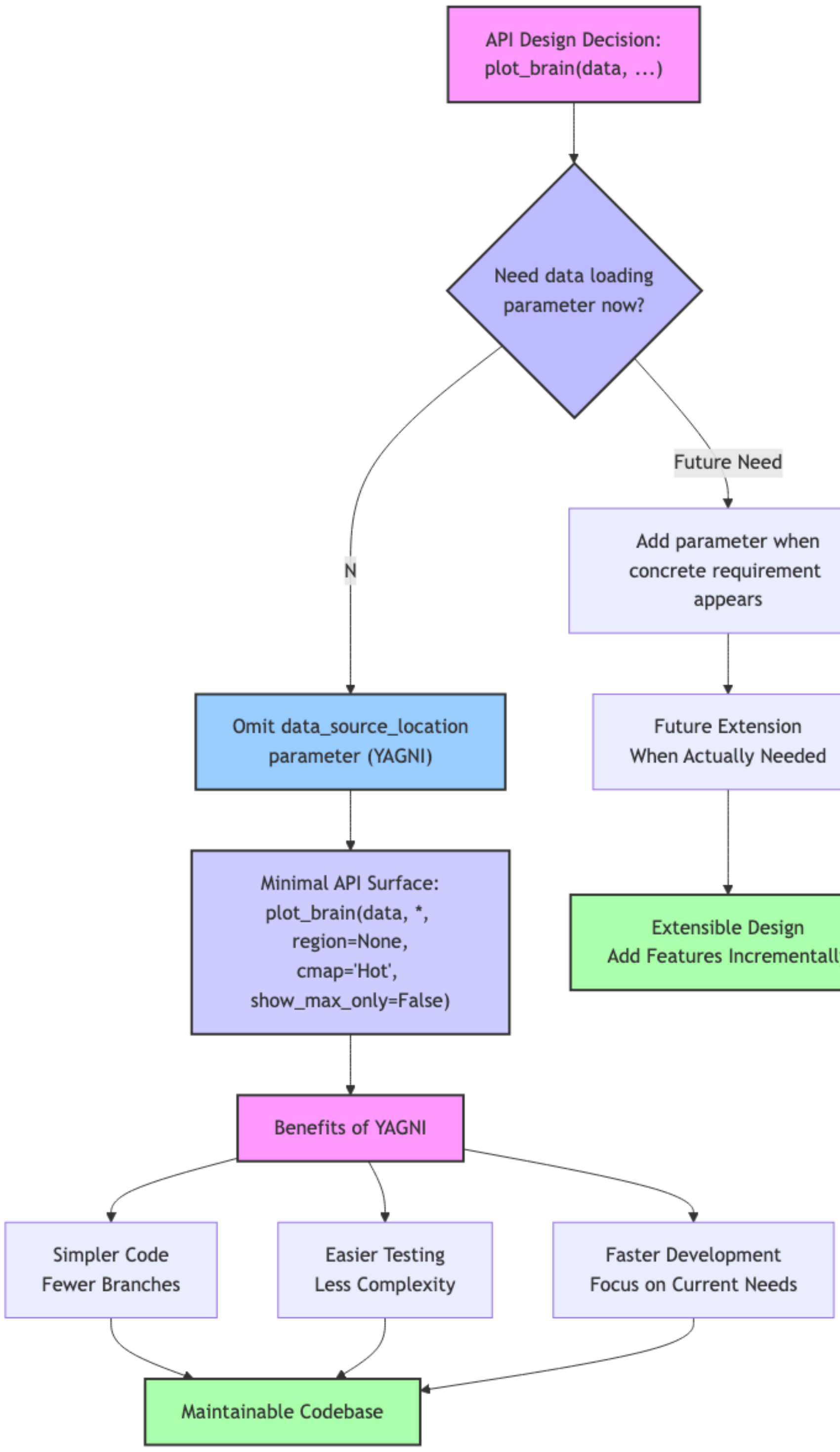
API Design Decision: plot_brain(data, ...)
Need data loading parameter now?
N
Future Need
Add parameter when concrete requirement appears
Omit data_source_location parameter (YAGNI)
Future Extension When Actually Needed
Minimal API Surface: plot_brain(data, *, region=None, cmap='Hot', show_max_only=False)
Extensible Design Add Features Incrementally
Benefits of YAGNI
Simpler Code Fewer Branches
Easier Testing Less Complexity
Faster Development Focus on Current Needs
Maintainable Codebase

Figure 3.2: YAGNI Principle: Minimal API Design Logic in LiveNeuron

3.3 The Pros and Cons of Design Principles

Adhering to design principles is generally considered a best practice that leads to higher-quality software. The primary benefit is that it makes systems more resilient to change, easier to debug, and more efficient for teams to collaborate on. By creating a clear and logical structure, these principles reduce the cognitive load on developers and help to manage the complexity of the system.

However, there can be trade-offs. A rigid adherence to design principles can sometimes lead to *over-engineering*, where the solution is more complex than the problem requires. This can increase the initial development time and effort. For example, creating a highly decoupled system might involve writing more "boilerplate" code to manage the communication between modules. The key is to apply these principles pragmatically, understanding that the goal is not to follow them dogmatically but to use them as tools to create a better, more sustainable software product.

The LiveNeuron development experience illustrates this pragmatic approach: the project began with tightly coupled code and evolved into a modular package only when concrete problems emerged—specifically, dependency conflicts between Plotly and Matplotlib that broke continuous integration tests. This iterative refinement proved more effective than attempting perfect architecture from the start. Similarly, while the DRY principle eliminated duplication in color scale computation, and YAGNI kept the API minimal, both principles were applied with flexibility: limited duplication remained acceptable when it improved scientific transparency, and the API preserved exploratory capabilities that researchers genuinely needed. Most importantly, when design principles conflicted with scientific requirements—such as the need for visual consistency across brain views—domain correctness took priority, with architectural solutions sought to minimize rather than eliminate the trade-off.

3.4 Case Study: The Evolution of LiveNeuron as a Modular Package

The development of the "LiveNeuron" package serves as a powerful case study in the practical application of software design principles. The initial approach was to develop the visualization tool as a new feature directly within the existing Eelbrain codebase. This was implemented in a single, large pull request that introduced thousands of lines of new code.

This initial strategy quickly revealed several challenges, which were documented in the code review process (see Chapter 4). First, the size of the pull request made it exceedingly difficult for reviewers to assess effectively, highlighting a **maintainability** issue. Second, and more critically, the new code introduced a dependency on the Plotly library, which created conflicts with the existing Matplotlib-based plotting tools in Eelbrain. This led to the failure of Continuous Integration (CI) tests, demonstrating a classic problem of tight coupling: changes in one part of the system caused unexpected failures in another.

The solution, which emerged from the review discussion, was to re-envision "LiveNeuron" not as an integrated feature, but as a separate, standalone Python package. This architectural shift was a direct application of the principles of modularity and decoupling. By moving the "LiveNeuron" code to its own repository, several key advantages were gained:

1. **Reduced Complexity:** The Eelbrain core remained unchanged and stable,

and the “LiveNeuron” package could focus solely on its specific visualization tasks.

2. **Dependency Isolation:** The conflict between Plotly and Matplotlib was resolved, as they now existed in separate, independent packages.

3. **Improved Maintainability:** The smaller, focused codebase of “LiveNeuron” was much easier to review, test, and maintain independently of the main Eelbrain project.

This case study demonstrates that while the initial goal was simply to add a feature, the challenges encountered during development and code review forced a reconsideration of the software’s design. The final decision to create a modular, decoupled package was not just a technical fix but a strategic design choice that improved the long-term health and maintainability of the entire software ecosystem.

This decision raises a practical question: when is a separate package feasible, and when is it not? In our experience, separation is feasible when there is a clear domain boundary and a stable interface to the core (e.g., NDVar in/out), the functionality is optional for many users (e.g., visualization), dependencies are heavy or conflicting (Plotly vs. Matplotlib), or the component benefits from an independent release cadence and ownership. It is less feasible when the feature is tightly coupled to evolving internals, cross-cuts many modules, is so small that split overhead outweighs benefits, or must meet hard performance or UX constraints that require it to “just be there” in the main API. Therefore, splitting should be a deliberate choice, not a default: it brings modularity and isolation at the cost of extra repos, versioning, CI, and user installation friction.

Concretely, a rule of thumb that guided the LiveNeuron split:

- **Split when**: optional feature, stable API boundary, heavy/conflicting deps, reuse outside core, or different cadence.
- **Keep in core when**: fundamental for most users, tightly coupled to changing internals, very small scope, or requires in-process performance/shared state.

3.5 Advice for Research Software Developers

This chapter has examined three fundamental design principles—SOLID (particularly LSP), DRY, and YAGNI—through concrete examples from the "LiveNeuron" visualization project. The analysis reveals several key insights specific to research software development contexts.

First, the application of the Liskov Substitution Principle demonstrated how input normalization enables flexible data handling while maintaining code simplicity. By ensuring that both vector and scalar NDVar inputs are processed through the same downstream logic, we achieved maintainability without sacrificing scientific functionality. This approach is particularly valuable in research contexts where data formats evolve frequently. *When this may not apply:* if downstream algorithms fundamentally depend on vector directionality versus scalar magnitude in ways that cannot be reconciled by normalization, forcing a single pathway would hide essential scientific distinctions.

Second, the DRY principle proved essential for maintaining consistency across multiple visualization views. Computing color scale limits once and sharing them across all brain views not only eliminated code duplication but also ensured visual coherence—a critical requirement for scientific visualization where comparisons between views must be meaningful. *When this may not apply:* if views serve different analytical purposes or distributions (e.g., raw vs. normalized metrics), enforcing one shared scale can reduce interpretability or mask local structure.

Third, YAGNI guided us toward minimal API design by avoiding speculative features like data loading parameters. This principle is especially important in research software where requirements often emerge iteratively, and premature feature addition can lead to maintenance burdens. *When this may not apply:* if a near-certain requirement would be prohibitively expensive to retrofit later (e.g., a data source selector that determines storage layout), a minimal anticipatory hook can be justified with clear rationale.

The LiveNeuron case study illustrates the practical consequences of these principles at the architectural level. The transition from a monolithic integration to a modular package structure resolved dependency conflicts and improved maintainability, demonstrating that modular design serves both technical and scientific objectives.

For research software developers, we recommend: (1) applying input normalization patterns early to accommodate evolving data formats, (2) using shared state management for visualization consistency, (3) keeping APIs minimal until concrete requirements emerge, and (4) considering modular architectures to prevent dependency conflicts between specialized scientific libraries. Most importantly, design principles should enhance rather than constrain scientific workflows—when conflicts arise, prioritize domain correctness while seeking design solutions that preserve both scientific validity and code maintainability.

Chapter 4

Code Review and Design Refinement

Software design principles, as discussed in Chapter 3, provide theoretical guidance for building maintainable and scalable systems. However, the translation of these principles into working code is rarely perfect on the first attempt. Code review serves as a critical mechanism for bridging the gap between design intent and implementation reality, offering opportunities to identify issues, refine approaches, and ensure alignment with established principles before changes become permanent parts of the codebase.

This chapter examines the role of code review in research software development, drawing on concrete experiences from the LiveNeuron project. Section 4.1 establishes the fundamental rationale for code review in scientific computing contexts, where correctness and reproducibility are paramount. Section 4.2 describes the specific code review practices employed in the Eelbrain project, including pull request workflows and continuous integration support. Section 4.3 presents a balanced analysis of the benefits and challenges associated with code review, acknowledging both its value and its costs in research environments (Eisty and Carver, 2022). Finally, Section 4.4 provides a detailed case study of how code review led to a significant performance optimization in the arrow visualization feature, demonstrating the tangible impact of the review process on software quality and user experience.

4.1 Why We Need Code Review

Code review is a critical practice in modern software development, yet its adoption and formalization can vary, especially in scientific computing. The primary motivation for code review is to improve the quality of the software. This involves not only identifying bugs and style issues but also verifying that the code correctly implements the desired methodology and produces reproducible results—a concern of paramount importance in science. As a 2022 empirical study by Eisty and Carver found, many research software teams perform code reviews but often lack a formal process, which can limit the effectiveness of the review (Eisty and Carver, 2022). The collaborative nature of code review also serves to disseminate knowledge throughout a team, helping to onboard new members and reduce dependency on any single developer. Furthermore, the process reinforces a culture of collective ownership and shared responsibility for the codebase, a topic emphasized in recent Better Scientific Software seminars (Kershaw, 2024).

4.2 Code Review Practices in Eelbrain Development

In the context of the Eelbrain project and the development of the “LiveNeuron” package, several code review methodologies were employed. The most common of these was the Pull Request (PR) workflow, facilitated by platforms like GitHub. This method allows for an asynchronous, lightweight peer review where contributors submit changes that can be examined alongside automated test results. This aligns with the findings of Eisty and Carver (2022), who noted that a majority of research developers initiate reviews through pull requests.

Continuous Integration (CI) and automated code analysis tools also support the review process (Fowler, 2006). By setting up CI (via GitHub Actions, Travis CI, etc.), teams ensure that every code change is automatically tested on a clean environment – this catches bugs early and provides confidence to reviewers that the code at least runs and passes basic tests. Linters and static analysis (like Pylint for Python) can automatically check code style and flag common errors, offloading some burden from human reviewers. A key principle of this workflow is that developers are not permitted to merge their own pull requests; all changes require at least one independent review before being integrated into the main branch.

4.3 Advantages and Disadvantages of Code Review

Code review, while widely recognized as a best practice in software development, presents both significant benefits and notable challenges, particularly in the context of research software development (Eisty and Carver, 2022).

4.3.1 Advantages of Code Review

Quality Improvement: Code review serves as a critical quality gate, catching bugs, logic errors, and edge cases that individual developers might overlook (Ahmad et al., 2023). In our "LiveNeuron" development experience, reviewers identified several critical issues including improper handling of different data formats and potential performance bottlenecks that would have affected end users[1].

Knowledge Sharing and Learning: The review process facilitates knowledge transfer across team members. In the Eelbrain project, reviews helped disseminate understanding of neurophysiological data structures and visualization requirements across the development team, as exemplified by discussions during the implementation of feature enhancements like real-time plot updates[2].

Design Validation: Reviews provide an opportunity to assess whether new code adheres to established design principles such as modularity, decoupling, and single responsibility. For instance, reviewers can identify when a pull request introduces unnecessary dependencies between components or when a function attempts to handle

[1]For example, a performance bottleneck for vector plot updates was identified and tracked in this GitHub issue: `https://github.com/liang-bo96/LiveNeuron/issues/2`

[2]See the pull request discussion for the implementation of this feature at `https://github.com/liang-bo96/LiveNeuron/pull/7`

too many responsibilities, as was the case in the initial attempt to integrate LiveNeuron into Eelbrain[3].

Consistency Enforcement: Code review helps maintain consistent coding standards, naming conventions, and architectural patterns across the entire codebase, making the software more maintainable and reducing the learning curve for new team members. For instance, in the development of LiveNeuron, a reviewer suggested renaming a parameter from `colorscale` to `cmap` to align with the existing Eelbrain API. This seemingly minor suggestion is critical for API usability, as consistent naming allows users to transfer knowledge between different functions more easily. The same review also enforced standards for the order of import statements and the format of docstrings, further contributing to a uniform codebase[4].

4.3.2 Disadvantages of Code Review

Development Delays: The review process can significantly slow down the development cycle, particularly when reviewers are unavailable or when complex changes require extensive discussion (Eisty and Carver, 2022). In research environments where rapid prototyping is often necessary, these delays can impede scientific progress.

Review Bottlenecks: Senior team members often become review bottlenecks, as their expertise is required for complex changes. This can create queues and further delays, especially in small research teams where domain expertise is concentrated in few individuals.

[3]The initial integration Pull Request (https://github.com/liang-bo96/Eelbrain/pull/16) was ultimately closed in favor of a separate package after review highlighted dependency conflicts and the large scope of the changes.

[4]These and other examples of consistency enforcement can be seen in the review discussion at `https://github.com/liang-bo96/Eelbrain/pull/16`

Subjectivity and Conflicts: Different reviewers may have conflicting opinions on implementation approaches, leading to lengthy discussions or inconsistent feedback. The subjective nature of some review comments can also create tension between team members if not handled diplomatically.

Review Fatigue: Large or complex pull requests may receive superficial reviews due to reviewer fatigue, potentially missing critical issues. Our experience showed that changes exceeding 300-400 lines of code often received less thorough examination.

False Security: Teams may develop over-reliance on the review process, potentially reducing individual responsibility for code quality and comprehensive testing before submission.

Challenges in Visual Software Review: A specific challenge arises in the review of scientific and image processing software, such as LiveNeuron. During a typical code walkthrough, it is difficult to visualize intermediate outputs—the images or plots generated at various stages of a data processing pipeline. Unlike purely algorithmic code where intermediate variable states can be inspected, the visual nature of the software's output makes step-through debugging and review less straightforward. This can make it challenging for reviewers to validate the correctness of each step in the visualization process.

The key to effective code review lies in balancing these trade-offs through appropriate process design, clear guidelines, and tool support that maximizes benefits while minimizing friction in the development workflow.

4.4 Case Study: Performance Optimization of Arrow Visualization

While asynchronous reviews via pull requests are effective for many aspects of software development, certain issues, particularly performance bottlenecks, are often best identified in a synchronous code review session, or "code walkthrough." During a code walkthrough, the author of the code presents it to one or more reviewers, explaining the logic and structure in real time. This interactive format allows for immediate questions and a deeper dive into complex sections of the code.

The performance issue with arrow visualization in LiveNeuron was discovered during such a code walkthrough. The initial implementation of the arrow plotting function was straightforward but inefficient, as it created a separate graphical object for each arrow.

4.4.1 The Problem: Slow Rendering of Vector Fields

An early implementation of a feature to visualize vector fields involved drawing a large number of arrows on a plot. During code review, it was discovered that this process was extremely slow when more than a few dozen arrows were present, leading to an unresponsive user interface. The root cause was the method of drawing: each arrow was being added to the figure as a separate, individual annotation object. While this approach is straightforward to implement for a small number of elements, it is computationally expensive and does not scale well, as each annotation carries significant overhead.

4.4.2 The Solution: Batch Processing for Arrow Rendering

The solution was to refactor the code to use a batch processing method. Instead of iterating and creating hundreds of individual objects, the new approach uses vectorized calculations with NumPy to determine the start and end points of all arrows at once. It then adds all arrow lines as a single 'Scatter' trace and all arrowheads as a second 'Scatter' trace. This dramatically reduces the number of objects the plotting library has to manage, resulting in a significant performance improvement.

Original Code

```
// For each vector to be drawn:
FOR each vector in vector_list:
  // Calculate start and end coordinates for one arrow
  start_point = get_vector_origin(vector)
  end_point = start_point + get_vector_direction(vector) * scale

  // Add a single, separate arrow object to the plot
  // This is inefficient as it creates many individual objects
  DRAW_ARROW(from: start_point, to: end_point)
ENDFOR
```

Refactored Code with Batching

```
FUNCTION draw_arrows_in_batch(vector_list):
  // 1. Prepare all data points in bulk using vectorized operations
  all_start_points = GET_ALL_ORIGINS(vector_list)
  all_end_points = all_start_points + GET_ALL_DIRECTIONS(vector_list) * scale

  // 2. Create a single data structure for all arrow lines
  // All line segments are stored together
  line_segments = PREPARE_LINE_SEGMENTS(all_start_points, all_end_points)

  // 3. Draw all arrow lines in a single, efficient operation
  DRAW_MULTIPLE_LINES(line_segments)

  // 4. Create a single data structure for all arrowheads
  arrowhead_markers = PREPARE_ARROWHEAD_MARKERS(all_end_points)

  // 5. Draw all arrowheads in a single, efficient operation
  DRAW_MULTIPLE_MARKERS(arrowhead_markers)
ENDFUNCTION
```

4.4.3 Guidance for Future Developers

This case study provides a crucial lesson for developing scientific visualization tools: always anticipate that the software will be used with large datasets, a recommendation also emphasized by Guo (2014). When implementing features that render many graphical elements, developers should prioritize vectorized or batched operations over simple iterative approaches. For plotting libraries like Plotly, creating a multitude of individual annotations or shapes is far less performant than drawing a single, complex trace. This principle of batching not only improves performance but also often leads to cleaner, more maintainable code.

Chapter 5

Usability Study and User Feedback

5.1 The Importance of Usability in Design

In the development of scientific software, the ultimate goal is to create tools that accelerate research and facilitate discovery. Usability, or the ease with which a user can effectively and efficiently achieve their goals with a piece of software, is fundamental to this objective. A tool may be powerful and algorithmically sophisticated, but if it is difficult to use, it will not be widely adopted. Poor usability can lead to user frustration, a steep learning curve, and a higher likelihood of errors, all of which hinder scientific progress. Therefore, focusing on usability is not a secondary concern but a core component of high-quality software design, ensuring that the software's capabilities are accessible to its intended audience of researchers.

5.2 The Impact of User Feedback on Design Modifications

The user feedback gathered during the development process had a direct and significant impact on the design of the "Live Neuron" feature. Several key modifications were implemented as a direct result of this expert user input. For example, one piece of feedback noted that the parameter for controlling the color map was named `colorscale`. To improve consistency with the rest of the Eelbrain library and the wider Python scientific ecosystem, this was changed to the more standard `cmap`.

Another critical set of changes involved the user interaction workflow. It was noted that to select a time point, a user had to click on a specific line in the time-series plot; this was updated to allow a click anywhere in the plot area, making the interaction more fluid. Similarly, initial versions displayed a separate color bar for each brain view, even when they showed the same data. The design was revised to use a single, shared color bar, making the visualization less cluttered and easier to interpret. These examples illustrate how direct, iterative user feedback can identify specific points of friction and lead to concrete improvements that enhance the user experience.

5.3 Integrating Usability Research into Future Design

The experience of improving the "Live Neuron" feature highlights the value of integrating user feedback throughout the development lifecycle. For future development in the Eelbrain ecosystem, a more continuous and formalized approach to usability research is recommended. This does not have to involve large-scale studies for every change, but can be integrated in more lightweight ways.

For instance, developers can engage in early-stage prototyping and wireframing, gathering feedback from a small group of users before significant coding effort is invested. Additionally, incorporating an in-application feedback mechanism could provide a continuous stream of user suggestions and pain points. By treating usability as an ongoing process rather than a one-time evaluation, the Eelbrain project can ensure that it continues to evolve in a user-centered direction, creating tools that are not only powerful but also a pleasure to use.

Chapter 6

Conclusion

This report has chronicled the journey of developing the "LiveNeuron" visualization package, using it as a case study to explore several critical facets of modern scientific software development. By examining the practical challenges encountered and the solutions implemented, this work has bridged the gap between abstract software engineering principles and their concrete application in a real-world research setting.

The discussion began with an exploration of fundamental **design principles**, demonstrating how an initial, tightly-coupled design led to significant integration and maintenance challenges. The evolution of "LiveNeuron" into a standalone, modular package underscored the importance of principles like the Single Responsibility Principle and the Law of Demeter in creating robust and maintainable software. This experience serves as a practical example of how architectural decisions, guided by established principles, can profoundly impact a project's long-term viability.

Subsequently, this report highlighted the indispensable role of **code review** not merely as a tool for catching bugs, but as a collaborative process for design refinement. The case study on performance optimization illustrated how peer review can identify critical issues that might otherwise go unnoticed, leading to a more efficient and scalable solution. Code review reinforces design principles and fosters a culture of shared ownership and continuous improvement.

The report also addressed the crucial, yet often overlooked, aspects of **contributor guidelines** and **usability**. The challenges in collaboration within the Eelbrain project pointed to the need for clear, comprehensive documentation to lower the barrier for new contributors. Similarly, the usability study demonstrated that direct user feedback is invaluable for creating tools that are not only powerful but also intuitive for the intended scientific audience.

In conclusion, the development of high-quality scientific software is a multi-faceted endeavor that extends far beyond just writing functional code. It requires a thoughtful approach to software design, a commitment to collaborative practices like code review, and a user-centered focus on usability. The lessons learned from the "LiveNeuron" project offer valuable insights for the scientific computing community, emphasizing that investing in these areas is essential for creating sustainable, impactful, and widely adopted research tools.

6.1 Summary of Guidelines for Research Software Development

Based on the case studies and analysis presented in this report, we offer the following guidelines for developers of research software:

1. **Prioritize Modularity and Isolate Dependencies**: To manage complexity and avoid maintenance issues, separate specialized tools from the core domain

library, especially when they introduce conflicting dependencies. This architectural decision, as demonstrated by the evolution of LiveNeuron, is crucial for long-term project health (see Section 3.4).

2. **Anticipate Large Datasets and Prioritize Performance**: In scientific visualization, always assume the software will handle large datasets. Prioritize vectorized or batched operations over simple loops from the outset to ensure performance, a lesson learned from the arrow visualization optimization (see Section 4.4).

3. **Use Different Code Review Formats for Different Goals**: Employ asynchronous pull request reviews for routine changes and enforcing standards, but use synchronous "code walkthroughs" to tackle complex design problems or performance issues that benefit from real-time, interactive discussion (see Chapter 4).

4. **Apply Design Principles Pragmatically**: Treat principles like DRY and YAGNI as tools, not rules. The goal is to create sustainable software, which sometimes requires balancing strict adherence to a principle against the practical needs of clarity or future extensibility (see Section 3.3).

5. **Establish Clear Contributor Guidelines**: To encourage collaboration and maintain code quality, create comprehensive contributor guidelines that cover coding standards, testing procedures, and architectural principles (see Chapter C).

Appendix A

LiveNeuron Usability Test Questionnaire

A.1 Introduction

Thank you for participating in the LiveNeuron usability test! LiveNeuron is a Python tool designed for interactive 2D visualization of EEG/MEG brain data. Your feedback is crucial for improving the user experience and making this tool more accessible to the neuroscience research community.

A.2 Part 1: User Background Information

1. What is your profession or field of study?

    - Neuroscience Researcher
    - Data Scientist

- Software Developer
- Graduate Student
- Undergraduate Student
- Postdoctoral Researcher
- Other: ______________

2. How long have you been using Python for data analysis or visualization?

   - Less than 6 months
   - 6 months to 1 year
   - 1-3 years
   - 3-5 years
   - More than 5 years

3. Have you used other EEG/MEG data visualization tools?

   - Yes
   - No

4. If yes, please list the tools you use most frequently:

____

5. How would you rate your overall experience with data visualization tools?

   - Beginner
   - Intermediate
   - Advanced
   - Expert

A.3 Part 2: Task-Based Evaluation

Please attempt the following tasks and answer the questions based on your experience. We recommend starting with the `example.py` script or the code examples in the README.

A.3.1 Task 1: Installation and First Run

Action: Follow the "Installation" and "Quick Start" sections in the `README.md` to install LiveNeuron and run the interactive dashboard with sample data.

6. Was the installation process clear and smooth?

    - Very smooth - completed without any issues
    - Mostly smooth - minor issues but manageable
    - Neutral - some confusion but eventually successful
    - Somewhat difficult - required troubleshooting
    - Very difficult - major issues encountered

7. Please describe any issues you encountered:

---

8. Were the instructions to launch the first visualization dashboard (`viz.run()`) easy to follow?

    - Very easy - immediately understood and executed
    - Easy - clear instructions with minimal confusion
    - Neutral - required some interpretation

- Somewhat difficult - needed additional research
- Very difficult - instructions were unclear

A.3.2 Task 2: Exploring the Interactive Visualization Interface

Action: In the running dashboard, explore the following features:

- Examine the three brain projection views (axial, sagittal, coronal)
- Click on different time points in the butterfly plot
- Observe how the brain projection views change when you interact with the butterfly plot

9. Is the interface layout (three brain views + butterfly plot) intuitive?
    - Very intuitive - immediately understood the layout
    - Mostly intuitive - quickly figured out the components
    - Neutral - took some time to understand
    - Slightly confusing - unclear relationships between components
    - Very confusing - difficult to understand the interface
10. How smooth was the interaction of selecting a time point on the butterfly plot to update the brain views?
    - Very smooth and responsive - immediate updates
    - Mostly smooth - quick updates with minimal delay

- Acceptable - noticeable but reasonable delay
- Laggy - slow updates affecting usability
- Unresponsive - often failed to update or very slow

11. Is the information presented clearly? (e.g., axes, color bars, time indicators)
    - Very clear - all elements well labeled and understandable
    - Mostly clear - minor ambiguities but generally good
    - Neutral - some elements clear, others less so
    - Somewhat unclear - several confusing elements
    - Very confusing - difficult to interpret the visualizations

A.3.3 Task 3: Customizing the Visualization

Action: Try the code from the "Advanced Usage" section in the `README.md` to perform the following:

- Change the colormap (`cmap`)
- Apply a brain parcellation (`region='aparc+aseg'`)
- Toggle the display mode of the butterfly plot (`show_max_only`)

12. Were the code parameters for customizing the visualization easy to understand and use?
    - Very easy - parameter names and usage were obvious
    - Easy - clear documentation and straightforward syntax

- Neutral - required some experimentation
- Somewhat difficult - unclear parameter effects
- Very difficult - confusing parameters and documentation

13. Do you find the provided customization options sufficient for your typical needs?

    - Completely sufficient - covers all my visualization needs
    - Mostly sufficient - covers most important use cases
    - Neutral - covers basic needs but missing some features
    - Somewhat insufficient - missing several important options
    - Completely insufficient - lacks many essential features

14. What other customization features would you like to see?

A.3.4 Task 4: Exporting Images

Action: Use the `viz.export_images()` function to export the visualization at a specific time point as static images.

15. Did the image export feature work as you expected?

    - Yes
    - No

16. Please describe any issues you encountered:

A.4 Part 3: Overall Impressions and Suggestions

A.4.1 Performance and Usability

17. What is your impression of LiveNeuron's performance (especially arrow rendering and interaction speed)?

    - Very fast - excellent performance throughout
    - Fast - good performance with minimal delays
    - Average - acceptable performance for most tasks
    - Slow - noticeable delays affecting workflow
    - Very slow - performance issues significantly impacted use

18. How would you rate the documentation (`README.md`)?

    - Excellent
    - Good
    - Fair
    - Poor

19. What areas of the documentation could be improved?

---

20. What do you think is the most valuable feature of LiveNeuron?

---

21. What was the biggest difficulty you faced while using the tool?

---

22. How likely are you to recommend LiveNeuron to a colleague or friend?

---

23. Do you have any other suggestions or ideas for improving LiveNeuron?

---

24. How was your user experience about Live Neuron?

Appendix B

Software Requirements Specification for LiveNeuron

B.1 Introduction

Electrophysiological data representing human brain activity is inherently complex and multidimensional, originating from numerous synchronized neural sources rather than individual neurons. Live Neuro, an interactive data visualization tool, is specifically designed to effectively present and explore this non-invasive electrophysiology data, helping users clearly understand the collective neural dynamics.

B.1.1 Purpose of Document

The following section provides an overview of the Software Requirements Specification (SRS) for the interactive neural data visualization tool. The purpose of this program is to serve as a basic data visualization foundation module in Eelbrain, making all neuro-related plots interactive within Jupyter notebook, which is not currently available in Eelbrain. This section explains the purpose of this document, the scope of the requirements, the characteristics of the intended reader, and the organization of the document.

B.1.2 Scope of Requirements

The scope of requirements include the interactive data visualization.

B.1.3 Characteristics of Intended Reader

Reviewers of this documentation should have an undergraduate-level understanding of math and neuro science.

B.1.4 Organization of Document

The organization of this document follows the template for an SRS for scientific computing software. The presentation follows the standard pattern of presenting goals, theories, definitions, and assumptions. For readers that would like a more bottom up approach, they can start reading the instance models and trace back to find any additional information they require.

B.2 General System Description

This section provides general information about the system. It identifies the interfaces between the system and its environment, describes the user characteristics and lists the system constraints.

B.2.1 System Context

Figure B.1 shows the system context. A circle represents an external entity outside the software, the user in this case. A rectangle represents the software system itself (Live Neuro). Arrows are used to show the data flow.

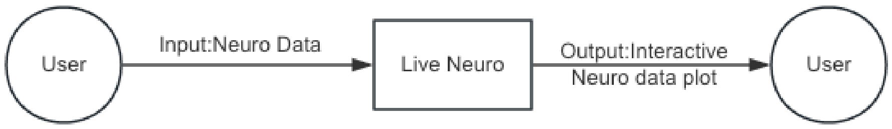

Figure B.1: System Context

- **User Responsibilities:**
  - Provide the input data to the system, ensuring no errors in the data entry.
- **Live Neuro Responsibilities:**
  - Detect data type mismatch, such as a string of characters instead of a floating point number.
  - Determine if the inputs satisfy the required software constraints, excluding invalid or corrupt data from analysis.
  - User continually provides input and the system has to incorporate that input in real time

B.2.2 User Characteristics

The end user of Live Neuro should have an understanding of Neuro Science including the basic neuro data structure.

B.2.3 System Constraints

The project should be developed under the plot module of Eelbrain, and the basic visualization fundamental Library should be Plotly or any other Library with interactive function in Jupyter notebook.

B.3 Specific System Description

This section first presents the problem description, which gives a high-level view of the problem to be solved. This is followed by the solution characteristics specification, which presents the assumptions, theories, and definitions that are used.

B.3.1 Problem Description

Live Neuro is intended to solve the problem of interactive neuro data visualization.

Terminology and Definitions

This subsection provides a list of terms that are used in the subsequent sections and their meaning, with the purpose of reducing ambiguity and making it easier to correctly understand the requirements. **dipole**: In neuroscience, a dipole refers to a pair of opposite electrical charges (positive and negative) that are separated by a small distance, typically generated by neuron activity (Gramfort et al., 2014).

Physical System Description

The physical system of Live Neuro, as shown in Figure B.2 (Das et al., 2020), includes the following elements: PS: Arrows in the Figure 2 are neuronal current dipole, representing the intensity of neuronal activity.

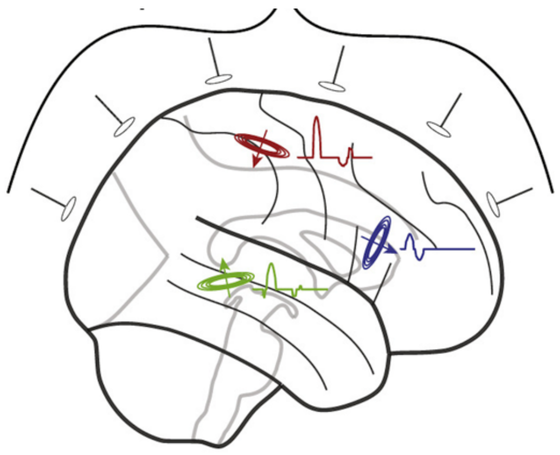

Figure B.2: Current Dipoles placed on the scalp

Goal Statements

Given the Neuro Data, the goal statements are:

GS1: Implement multidimensional neural data visualization.

GS2: Link Interactive neural data plots together.

B.3.2 Solution Characteristics Specification

Assumptions

A: The neuron data is complete.

Theoretical Model

Electromagnetic source imaging assumes a linear forward model that links brain source activity to sensor measurements. In mathematical form, one can write the forward model as:

$$\mathbf{y}(t) = \mathbf{L}\,\mathbf{j}(t) + \mathbf{n}(t)$$

where $\mathbf{y}(t)$ is the vector of measured signals at the sensors (EEG electrodes or MEG magnetometers (Gramfort et al., 2013)) at time t, $\mathbf{j}(t)$ is the vector of source currents (dipole moments) at each brain location (voxel or source grid point), and $\mathbf{L}$ is the lead field matrix (also called the gain matrix) that encodes the contribution of each unit source to each sensor. $\mathbf{n}(t)$ represents measurement noise.

Reconstructing brain activity at each voxel from sensor data requires solving the inverse problem, which is ill-posed (infinitely many source distributions can explain the same sensor measurements). To obtain a unique solution, additional constraints or priors are imposed, yielding an inverse model (a method to estimate source currents $\hat{\mathbf{j}}$ from $\mathbf{y}$). In practice, one defines an inverse operator (often denoted $\mathbf{W}$ or $\mathbf{M}$) such that:

$$\hat{\mathbf{j}}(t) = \mathbf{M}\,\mathbf{y}(t)$$

Minimum-Norm Estimation (Linear Inverse Solution).

Minimum-norm estimation assumes that out of all source configurations that explain the data, In a Bayesian interpretation, one assumes a Gaussian prior on source amplitudes favoring smaller values, and Gaussian sensor noise. The result is a regularized linear inverse operator $\mathbf{M}$ (of size [#sources $\times$ #sensors]) that can be expressed as:

$$\mathbf{M} = \mathbf{R}\,\mathbf{L}^T\big(\mathbf{L}\,\mathbf{R}\,\mathbf{L}^T + \lambda^2\mathbf{C}\big)^{-1}$$

where **L** is the lead field, **C** is the noise covariance matrix, **R** is the source covariance (source prior) matrix, and λ is a regularization hyperparameter controlling the trade-off between fidelity to the data and the norm of the sources.

B.4 Requirements

B.4.1 Functional Requirements Via Use Cases

FR1: The system shall allow users to render and interact with multidimensional neural plots.

FR2: The system shall highlight corresponding data points across multiple plots upon user interaction.

FR3: The system shall provide an option to export visualizations.

FR4: The system shall be usable within a Jupyter Notebook.

B.4.2 Nonfunctional Requirements

NFR1 - Scientific Visualization Quality: The generated data visualizations should not only accurately represent the underlying data but also adhere to the standards of scientific rigor, clarity, reproducibility, and interpretability. They must effectively convey key patterns, relationships, and trends in a manner that facilitates meaningful analysis and supports evidence-based conclusions.

NFR2 - Usability for Domain Experts: Users with knowledge of neuroscience and computer science should be able to successfully use the software with minimal training.

NFR3 - Modifiability Efficiency: The effort required to make any of the likely changes listed for Live Neuro should be less than 20% of the original development time.

NFR4 - Cross-Platform Compatibility: LiveNeuro shall run on Linux, Windows 10+, and macOS operating systems.

B.5 Likely Changes

The interactive method (mouse clicking, dragging, zooming etc.) may be modified or added according to the real needs.

The supported plotting function(3D Glassbrain function) may increase.

B.6 Unlikely Changes

The underlying source localization algorithm will not change.

Appendix C

Contributor Guide

C.1 Why a Contributor Guide is Necessary

A contributor guide is a critical document for any open-source software project, particularly in a scientific context like Eelbrain. Its primary purpose is to standardize the development process and lower the barrier to entry for new contributors. For a project to thrive and grow, it must be able to attract and effectively integrate new developers, who may range from students to researchers at other institutions.

A well-written guide provides a clear roadmap for everything from setting up a development environment and running tests to following coding conventions and submitting code for review. This standardization ensures that all contributions maintain a consistent level of quality, making the codebase more coherent and easier to manage in the long term. By removing ambiguity and providing clear instructions, a contributor guide empowers potential collaborators, reduces the burden on core developers to answer repetitive questions, and fosters a welcoming and productive community.

C.2 Recommendations: A Development Guide for Future Contributors

To address these challenges and support the future growth of Eelbrain, creating a comprehensive contributor guide is a crucial next step. This guide should be a living document, accessible in the project's main repository, and should cover several key areas:

C.2.1 Coding Conventions

To maintain a consistent and readable codebase, we adhere to the following conventions, which were established and refined through code review discussions.

- **Style Guide:** All Python code must adhere to the PEP 8 style guide. We use `flake8` to check for compliance. Before submitting your code, please run `flake8` locally and fix any reported issues. You can use tools like `autopep8` to automatically fix many common style issues.
- **Import Order:** Imports should be grouped in the following order, with a blank line separating each group:
  1. Standard library imports (e.g., `io`, `os`).
  2. Related third-party imports (e.g., `numpy`, `matplotlib`).
  3. Local application/library specific imports (e.g., `from eelbrain import` ...).

  Case Study: In one review, a contributor was reminded to reorder their imports to place standard library modules before third-party and local imports. Following this guideline makes it easier to see the dependencies of a module at a glance.

- **API Consistency:** To make the library intuitive, we strive for consistency across the API.
  - Parameter names should be consistent with existing functions. For example, use `cmap` for colormaps, not `colorscale`.
  - Functions that handle data should, where possible, accept data directly as a parameter (e.g., a `y` parameter for an `NDVar`), analogous to existing plotting functions.

    Case Study: During the development of the interactive plotting feature, a reviewer noted that a new function used the parameter name `colorscale`. The contributor was asked to rename it to `cmap` to match the parameter name used for colormaps throughout the rest of the Eelbrain library.

- **Type Hinting:** Use type hints in all function signatures (e.g., `def my_function(y: NDVar) -> Figure:`). When type hints are present in the signature, they should be omitted from the docstring to avoid redundancy.

- **Docstrings:** All public functions and classes should have clear and informative docstrings. Documentation for a class's `__init__` method should be included in the main class docstring.

- **Data Access:** When working with `NDVar` objects, use the `.get_data()` method to ensure the data axes are in the desired order, as the internal order is not guaranteed.

- **TODOs:** For in-code reminders, use a `TODO:` tag. For more significant or non-localized tasks, please open a GitHub issue instead.
- **Development Tools:** To streamline the development process and maintain code quality:
  - Use `flake8` locally to check code compliance before submitting
  - Consider using `autopep8` to automatically fix common style issues
  - Configure your IDE to automatically handle formatting (e.g., PyCharm can manage whitespace issues automatically)
  - Run local checks to catch style problems before they appear in CI
- **API Design Principles:** When designing new functionality:
  - Maintain parameter naming consistency across the codebase (e.g., use `cmap` for colormaps, not `colorscale`)
  - Follow existing patterns for data input (e.g., accept data directly via a `y` parameter like other plotting functions)
  - Consider usability in Jupyter environments during design
  - Document expected data formats clearly for users

C.2.2 Architecture and Dependency Guidelines (Learned from LiveNeuron)

To keep contributions sustainable and reviews effective, follow these project-level guidelines:

- **Reduce complexity via modular boundaries**: keep the Eelbrain core unchanged when possible; place specialized visualization or UI features in separate packages.
- **Isolate dependencies**: avoid mixing incompatible stacks (e.g., Plotly vs. Matplotlib) within one module; separate packages prevent CI failures and version conflicts.
- **Prefer smaller, focused units**: submit smaller repositories/modules and small PRs to simplify review, testing, and independent evolution.

C.2.3 The Pull Request (PR) Workflow

We use a pull request-based workflow for all contributions.

1. **Create a branch:** Create a new branch from `main` for your feature or bugfix.
2. **Make your changes:** Make your code changes, ensuring you follow the coding conventions.
3. **Submit a Pull Request:** When your changes are ready, push your branch to your fork and open a pull request against the `main` branch of the official Eelbrain repository.
    - Provide a clear and descriptive title for your PR.
    - In the description, explain the purpose of your changes and link to any relevant GitHub issues.
4. **Keep PRs small:** Whenever possible, break down large features into smaller, logically distinct pull requests. Small PRs are much easier and faster to review.

Case Study: The initial implementation of the "Live Neuron" feature was submitted as a single large PR (+1,873 lines of code). Reviewers noted that this made the review process very challenging. This experience highlighted the importance of smaller, more focused PRs to facilitate timely and effective feedback.

5. **Address feedback:** Engage with the code review process by responding to comments and pushing new commits to your branch to address the feedback.

C.2.4 Testing and Validation

Eelbrain relies on a robust test suite to ensure the correctness and reliability of its scientific algorithms.

- **Automated Testing:** All pull requests trigger a Continuous Integration (CI) workflow that automatically runs the full test suite. Please ensure all tests are passing before requesting a review.

  Case Study: The large "Live Neuron" PR initially failed the automated tests because a new dependency (Plotly) created conflicts with the existing Matplotlib-based code. This demonstrated the value of CI in catching integration issues early. The eventual solution was to develop the new feature as a separate, independent module to resolve the conflict.

- **Writing New Tests:** Any new feature or bugfix should be accompanied by corresponding unit tests. This helps prevent future regressions and validates that your code is working as expected.

- **Integration Testing:** For new modules (especially those with external dependencies like Plotly), ensure they:
  - Are compatible with the Jupyter notebook environment
  - Do not conflict with existing dependencies (particularly Matplotlib)
  - Work across supported platforms (Linux, Windows 10+, macOS)
  - Meet scientific visualization quality standards for accuracy and reproducibility
- **Performance Considerations:** When implementing interactive features:
  - Test with realistic data sizes and ensure reasonable response times
  - Consider memory usage for large datasets
  - Optimize for common use cases (e.g., showing important dipoles rather than all dipoles)
  - Verify that plot sizing works correctly within Jupyter cell outputs

C.2.5 Code Review Process

The code review process is a collaborative effort to improve the quality of the codebase.

- **What to Expect:** A core developer will review your pull request and may provide feedback on various aspects, including correctness, adherence to coding standards, API design, and usability. The process is iterative; you'll be expected to update your PR based on the feedback.

- **Applying Feedback:** When you receive feedback (e.g., about import order or type hints), please check your entire contribution to see if the same feedback applies elsewhere. This helps streamline the review process.

  Case Study: A reviewer provided feedback on the import order in one file. Later in the same review, the same issue was found in another file. This led to a friendly reminder to contributors to apply feedback globally across their entire PR, which makes the review more efficient for everyone.

- **Goal:** The goal of the review is not just to find errors but to refine the design and ensure the new code is well-integrated into the existing project. Once the core technical aspects are settled, we may also seek feedback from other users on the usability of a new feature.

- **Common Review Focus Areas:** Based on actual code review experiences, reviewers typically examine:
  - API consistency with existing functions (parameter names, data input patterns)
  - Code organization and import structure
  - Documentation completeness and clarity (including links to relevant external documentation)
  - Error handling for edge cases (e.g., data without certain dimensions)
  - User experience considerations (layout compactness, interaction patterns)
  - Performance implications for large datasets

- **Response to Feedback:** When addressing reviewer comments:
  - Apply feedback comprehensively across your entire contribution
  - Ask clarifying questions if requirements are unclear
  - Consider creating follow-up PRs for related improvements suggested during review
  - Document any design decisions that may not be immediately obvious

C.2.6 Scientific Software Requirements

As a scientific computing library, Eelbrain has additional requirements beyond typical software projects:

- **Reproducibility:** All visualizations and computational results must be reproducible across different platforms and Python environments. Code should produce consistent results given the same input data and parameters.
- **Scientific Accuracy:** Visualizations must accurately represent the underlying neuroscience data without introducing artifacts or misleading interpretations. This includes proper handling of coordinate systems, color scales, and data transformations.
- **Domain Compatibility:** New features should integrate seamlessly with the neuroscience workflow:
  - Support standard data formats (NDVar objects with appropriate dimensions)
  - Work within Jupyter notebook environments commonly used by researchers

  - Provide clear documentation of data expectations and output formats
  - Consider the typical use cases of neuroscience researchers

- **Performance for Research Data:** Neuroscience datasets can be large and complex:
  - Test with realistic data sizes (multiple subjects, high temporal resolution)
  - Optimize for common operations (time series visualization, source localization displays)
  - Provide user control over performance trade-offs (e.g., showing all vs. significant dipoles)

By investing in such a guide, the Eelbrain project can significantly improve its collaborative development process, making it more efficient, inclusive, and sustainable.

Bibliography

Israr Ahmad, Shahbaz Ali, and Pham Nguyen Huy. 2023. The Effectiveness of Code Reviews on Improving Software Quality: An Empirical Study. *International Journal of Computer Science and Network Security* 23, 7 (2023), 71–78.

Victor R Basili, Jeffrey C Carver, Daniela Cruzes, Lorin M Hochstein, Jeffrey K Hollingsworth, Forrest Shull, and Marvin V Zelkowitz. 2008. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE software* 25, 4 (2008), 29–36.

Christian Brodbeck. 2014. *Eelbrain: A Python toolkit for time-continuous analysis with temporal response functions.* `https://eelbrain.readthedocs.io/` Version 0.40.

Christian Brodbeck, L Elliot Hong, and Jonathan Z Simon. 2018. Rapid transformation from auditory to linguistic representations of continuous speech. *Current Biology* 28, 24 (2018), 3976–3983.

Jeffrey C Carver, Richard P Kendall, Susan E Squires, and Douglass E Post. 2007. Software development environments for scientific and engineering software: A series of case studies. *29th International Conference on Software Engineering (ICSE'07)* (2007), 550–559.

Proloy Das, Christian Brodbeck, Jonathan Z Simon, and Behtash Babadi. 2020. Neuro-current response functions: A unified approach to MEG source analysis under the continuous stimuli paradigm. *NeuroImage* 211 (2020), 116528.

Nibir Eisty and Jeffrey C Carver. 2022. An empirical study of code review practices for research software. *PeerJ Computer Science* 8 (2022), e958.

Martin Fowler. 2006. Continuous Integration. Online article. `https://martinfowler.com/articles/continuousIntegration.html`

Alexandre Gramfort, Martin Luessi, Eric Larson, Daniel A Engemann, Denis Strohmeier, Christian Brodbeck, Roman Goj, Mainak Jas, T Brooks, Lauri Parkkonen, and Matti S Hämäläinen. 2013. MEG and EEG data analysis with MNE-Python. *Frontiers in neuroscience* 7 (2013), 267.

Alexandre Gramfort, Martin Luessi, Eric Larson, Daniel A Engemann, Denis Strohmeier, Christian Brodbeck, Lauri Parkkonen, and Matti S Hämäläinen. 2014. MNE software for processing MEG and EEG data. *Neuroimage* 86 (2014), 446–460.

Philip J Guo. 2014. A practical framework for small teams to develop sustainable research software. *arXiv preprint arXiv:1403.2959*.

Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. How do scientists develop and use scientific software? *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering* (2009), 1–8.

Andrew Hunt and David Thomas. 1999. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional.

Rafael C Jiménez, Mateusz Kuzak, Monther Alhamdoosh, Michelle Barker, Bérénice Batut, Mikael Borg, Salvador Capella-Gutierrez, Neil Chue Hong, Martin Cook, Manuel Corpas, et al. 2017. Four simple recommendations to encourage best practices in research software. *F1000Research* 6 (2017).

Upulee Kanewala and James M Bieman. 2014. Testing scientific software: A systematic literature review. *Information and software technology* 56, 10 (2014), 1219–1232.

Helen Kershaw. 2024. Code Review for Scientific Software: Experiences building an online tutorial. Webinar, IDEAS Productivity Project. `https://ideas-productivity.org/events/hpcbp-082-codereview`

Robert C Martin. 2000. Design principles and design patterns. *Object Mentor* 1, 34 (2000), 597.

Marian Petre and Greg Wilson. 2014. Code review for and by scientists. *arXiv preprint arXiv:1407.5648* (2014).

Judith Segal. 2005. When software engineers met research scientists: A case study. *Empirical Software Engineering* 10, 4 (2005), 517–536.

Judith Segal and Chris Morris. 2008. Developing scientific software. In *2008 IEEE International Conference on Software Science, Technology and Engineering*. IEEE, 18–26.

Spencer Smith, Jacques Carette, and Nirmitha Koothoor. 2016. Software engineering for computational science: Past, present, future. *Computing in Science & Engineering* 18, 2 (2016), 90–99.

Tim Storer. 2017. Bridging the chasm: A survey of software engineering practice in scientific programming. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–32.

Greg Wilson, DA Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven HD Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, et al. 2014. Best practices for scientific computing. *PLoS biology* 12, 1 (2014), e1001745.

Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K Teal. 2017. Good enough practices in scientific computing. *PLoS computational biology* 13, 6 (2017), e1005510.