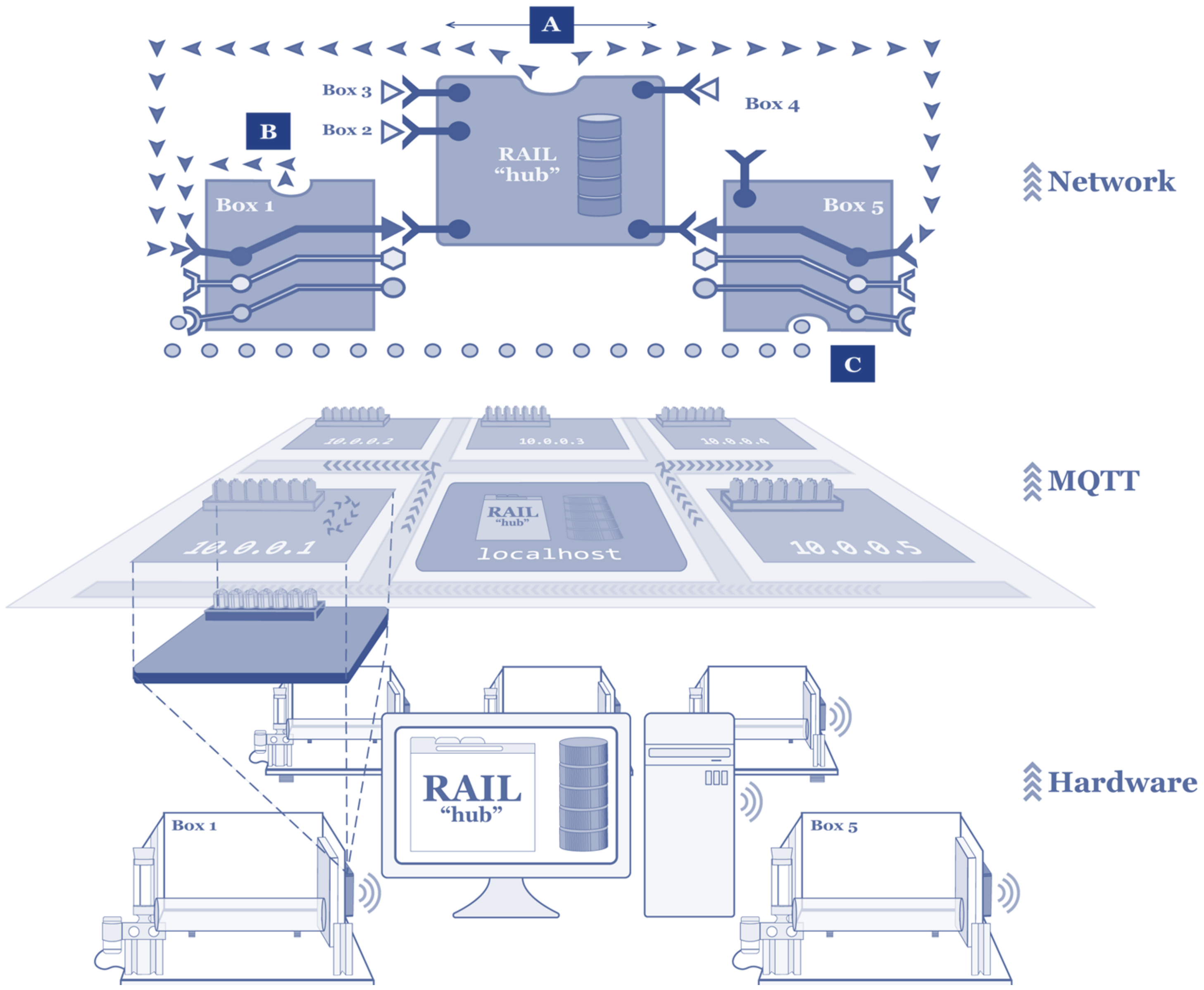


RAIL

Free and Open Source Software Project Fund (FOSSProF) Final Report

Barbara Holt

Mysore Lab

ABOUT RAIL

Code Repository: github.com/badholt/rail

The Rodent Automated and Integrated Learning (RAIL) system is an open source project leveraging web-based developer tools and open source software to operate Internet of Things (IoT) hardware in the behavioral training of animal subjects for neuroscience experiments, specifically visually guided tasks in mice. Our primary objective is to provide an open source community-based alternative for high-throughput operant behavioral training that pairs the user-friendly benefits of more costly commercial software systems with the specially tailored benefits of a home-grown system developed in the lab. By providing neuroscience researchers with a modular, versatile, and cost-effective platform for behavioral training, we hope to cultivate new research opportunities and community dialogue by lowering some of the financial and technical barriers to entry.

The average behavioral training paradigm may take many months to set up and implement and often requires engineering or technical expertise that lies outside the scope of a laboratory's research focus. Behavioral training outcomes also involve a considerable amount of animal subject attrition due to some animals failing to learn a particular task or complications arising from other aspects of the project, such as surgeries. Therefore, it is essential for researchers to be able to set up a novel training paradigm in a timely manner while simultaneously preparing to train as many subjects in parallel as possible.

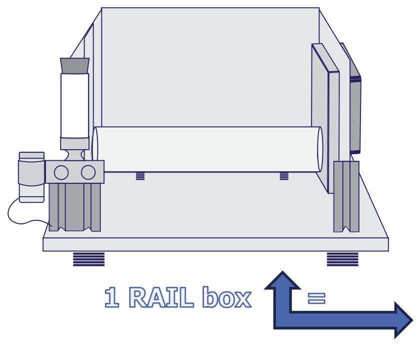
Commercial behavioral training systems provide a reliable user experience for researchers, usually offering robust technical support and detailed documentation of their systems. Unfortunately, in many cases, these commercial systems are also prohibitively expensive, with much of their proprietary software either "paywalled" or restricted in its functionality. Such financial limitations can significantly constrict the size of a research project, limiting a laboratory's ability to engage in massively parallel experiments.

Conversely, lab-grown systems feature highly customized and tailored designs, precisely targeted towards lab-specific methodologies and paradigms. However, these systems are also subject to varied reliability and burden a behavioral-focused lab with labor-intensive technical development and support. RAIL serves as a middle-ground option with an extensible modular and user-friendly design, open source community support, and added reliability and quality assurance thanks to Mysore Lab's extensive calibration, characterization, and beta testing. While RAIL setup requires individuals and labs to take a more active role in the experiment setup process than commercially available options, it does so for a fraction of the cost. Likewise, while unmodified RAIL software will not perfectly address every possible experimental paradigm, it provides a stable foundation for implementing novel use cases in a fraction of the time required for a lab-grown system. Ultimately, a cost-effective and user-friendly system for behavioral training permits neuroscience laboratories to focus more on research than setup by more easily generating a large pool of highly trained mice for massively parallel experiments.

ACTIVITIES & PROGRESS

Due to unanticipated delays in Johns Hopkins' procurement process, our scaling optimization efforts are still ongoing for the high-throughput version of the RAIL system. However, the months long process has allowed us to make significant inroads with the Node-based open source community, refine our internal beta testing and development processes, and focus on improving reliability and convenience for the end-user.

Through consultation with Meteor JS, MongoDB, and ScaleGrid engineers, we learned how to characterize our active database and project future data growth according to current industry standards. As a result, we were able to translate the prototypical usage of a RAIL operant training box into common database metrics, such as network and memory requirements. In the coming year, this per-box data equivalency unit will allow us to reinterpret code efficiency and database performance improvements in terms of added research capability and overall cost savings. Thus, we can now also provide higher fidelity cost estimates to potential users in the neuroscience and open source communities, as well as others who are interested in using or hosting RAIL software.



Network Bandwidth Requirements

Average Document Size	7725.525183
Objects per Second per Box	0.154
Documents per Second Served	4

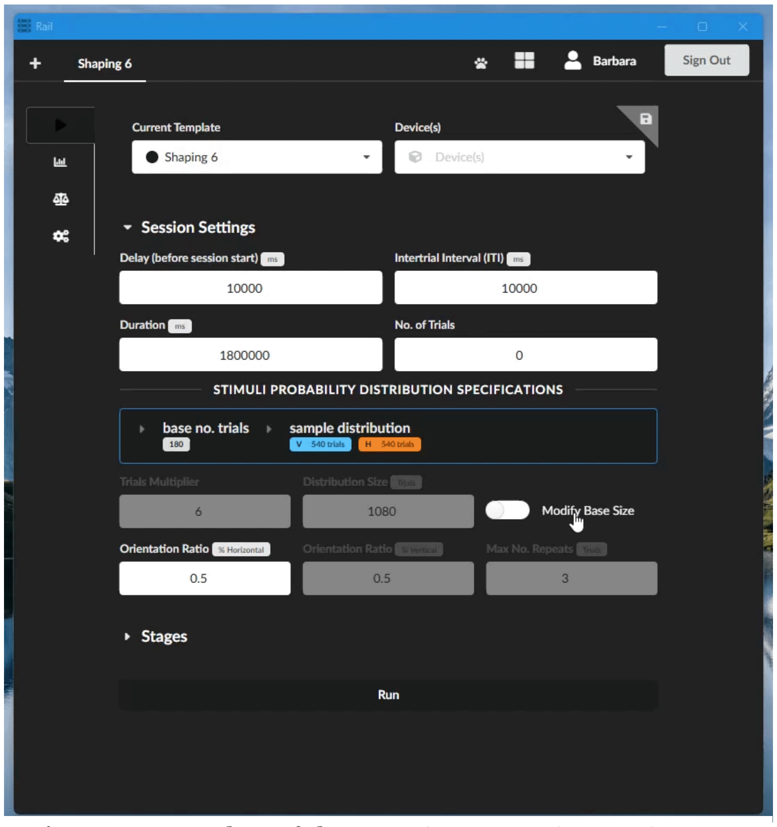
Fig. 1. Sample data equivalency for the network bandwidth requirements of one RAIL box running a Mysore Lab behavioral training paradigm of a visual discrimination task in mice.

In response to the logistical challenges involved in launching our scaling efforts, we pivoted our approach to emphasize more in-house beta testing of the RAIL system. We branched our code base into three separate versions of RAIL – (1) a master branch dedicated to "production," or in this case, established session operations, (2) a development branch for testing new features and bug fixes on the already established foundation, and (3) a desktop-capable version of RAIL which was updated to Meteor 3.0 and is no longer backwards compatible with previous versions of the software. In parallel, using MongoDB Compass and a series of custom installers, we also developed a new workflow for modifying the database as it underwent active development changes.

THE DEVELOPMENT BRANCH

From a behavioral standpoint, one of the most significant new features added to the RAIL development branch was the capacity to generate customizable *probability distributions*. Visual discrimination paradigms frequently include multiple stimuli or variants of a single visual stimulus. This requires the software to have a method of deciding which stimulus is shown when and for how often. For example, in earlier phases of Mysore Lab's standard visual discrimination task, a gratings stimulus will be shown oriented vertically for approximately half of an experiment session's trials and horizontally for the other half of an experiment session's trials. In the past, the order and probability of one stimulus orientation or the other occurring was handled by an algorithm with a random number generator (RNG). Expanding upon this basic functionality, several variables were exposed and incorporated into the graphical user interface (GUI) for users to customize, and an entirely new algorithm was developed to produce variable and weighted trial combinations beyond the standard 1:1 ratio.

Now, when researchers need to distribute the value of a particular variable across a single session, they may carefully control the proportions of individual outcomes. A default distribution is automatically calculated according to the default



Current Template: Shaping 6

Device(s): Device(s)

Session Settings

Delay (before session start) ms: 10000

Intertrial Interval (ITI) ms: 10000

Duration ms: 1800000

No. of Trials: 0

STIMULI PROBABILITY DISTRIBUTION SPECIFICATIONS

base no. trials: 180

sample distribution: V 540 trials, H 540 trials

Trials Multiplier: 6

Distribution Size Trials: 1080

Modify Base Size

Orientation Ratio % Horizontal: 0.5

Orientation Ratio % Vertical: 0.5

Max No. Repeats Trials: 3

Stages

Fig. 2. Screenshot of the experiment session settings GUI in web-based RAIL application. Here, a template for Shaping 6, a late-stage Mysore Lab visual task paradigm, is loaded and awaiting modification of the probabilities of either a horizontal or a vertical gratings stimulus occurring in an upcoming Shaping 6 session.

values of a given experiment template. However, the overall distribution size may either be adjusted relative to this default size or updated to an absolute amount. Next, the desired percentage of each stimulus variant may be specified, with a readout on the GUI reporting a preview calculation of the number of trials presenting each variant.

Finally, upon session initiation, a session scaffold is generated using the modified template, and the generated trials are shuffled with an efficient O(*n*) Fisher-Yates shuffling algorithm. The experiment session will then proceed either for a specified number of trials or a specified duration in milliseconds. However, independent of the number of trials initiated, the probability of a given stimulus variant being presented will be determined by the researcher's specified probability parameters. The percentages and distribution size will essentially determine the number of each type of trial in the probability "bag," while interactions from the mouse will determine which trials are presented, as if the mouse were drawing "tiles" from the probability bag. In addition to the presentation of a specific stimulus, other session variables may utilize this same probability distribution functionality across trials. For example, the opacity or location of a stimulus may vary according to a preset holistic probability distribution.

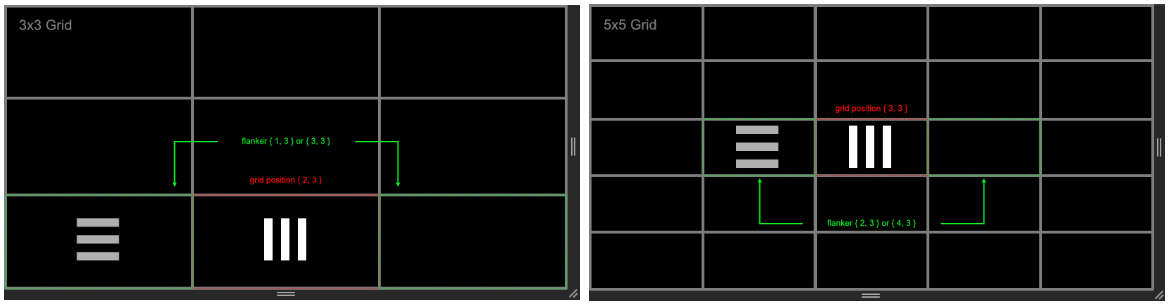
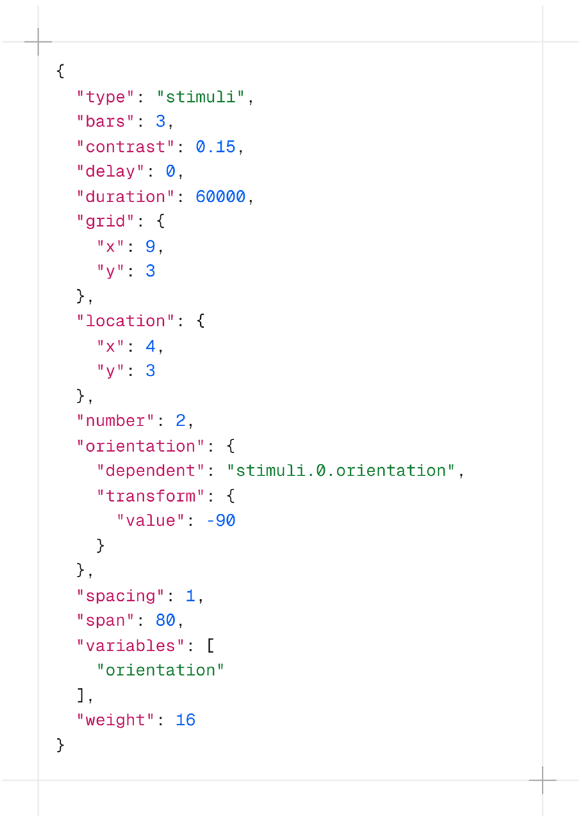


Fig. 3. Locations of two gratings stimuli from Mysore Lab's late-stage Shaping 8A visual paradigm are shown relative to a 3x3 and a 5x5 calibration grid on a 800x400 px screen. The text overlays shown were for labeling purposes only and are not present in the RAIL GUI.

When multiple stimuli and variables are distributed across a session, the combinatorial probabilities can become quite complex. Thus, a second major update to RAIL involved the introduction of modular template definitions for dependent variables. In other words, the value of one variable, such as a stimulus' location on a grid, may now be described relative to another variable, such as a second stimulus' location on the grid, within our portable templates. Previously, dependent variables were hard-coded within custom functions. Although new custom functions and methods could be added to RAIL, on a practical level, this rendered our software more lab-specific, tailored to the idiosyncrasies of Mysore Lab operations. The addition of this expansive capability supports community usage by exponentially increasing the diversity of possible relationships between experiment components. For example, a stimulus' opacity may now be linked to its orientation, or the screen brightness



```
{
  "type": "stimuli",
  "bars": 3,
  "contrast": 0.15,
  "delay": 0,
  "duration": 60000,
  "grid": {
    "x": 9,
    "y": 3
  },
  "location": {
    "x": 4,
    "y": 3
  },
  "number": 2,
  "orientation": {
    "dependent": "stimuli.0.orientation",
    "transform": {
      "value": -90
    }
  },
  "spacing": 1,
  "span": 80,
  "variables": [
    "orientation"
  ],
  "weight": 16
}
```

Fig. 4. A sample JSON object describing a flanker stimulus similar to the grey gratings shown in Figure 3. For this flanker stimulus, the orientation of the gratings is dependent on the orientation of the target stimulus. Here, the 'transform' property specifies that the flanker orientation will always be rotated 90° clockwise, or perpendicular to the target stimulus.

may be linked to the duration of an audio tone. By moving beyond the need for dependent variables to be enveloped in hard-coded functions, RAIL software can now naively reproduce more experimental paradigms and implementations straight "out of the box" with shareable JSON templates.

Among several bug fixes to templates and user accounts, we also added a third major enhancement to the user experience – a calibration view. As stimulus presentation and reward delivery are critical for behavioral training on visual tasks, researchers must ensure that stimuli are presented in their proper locations and that water rewards are delivered in precise amounts at all times. However, economical IoT hardware typically does not deliver the same degree of precision and performance as more expensive commercial products. This may result in a significant increase in the amount of time a researcher spends calibrating component devices. Thus, in order to minimize the time spent on calibration and daily session operations, we've added a new "calibration view" for stimulus alignment on the screen, as well as for modification of the milliliters of water reward delivered according to a customizable linear function. In both cases, to maximize convenience for the researcher, calibration values for individual boxes are specified as offsets instead of absolute values. This allows all operable boxes or devices to be run *en masse,* initiated simultaneously from a single session template but each delivering water reward and visuals according to their individual calibration requirements. In principle, calibration offsets may be applied to the characteristics of any hardware component, such as an alternative liquid or food reward delivery system. In a future code release, we plan on updating this language and our documentation to reflect a more generic usage of this feature.

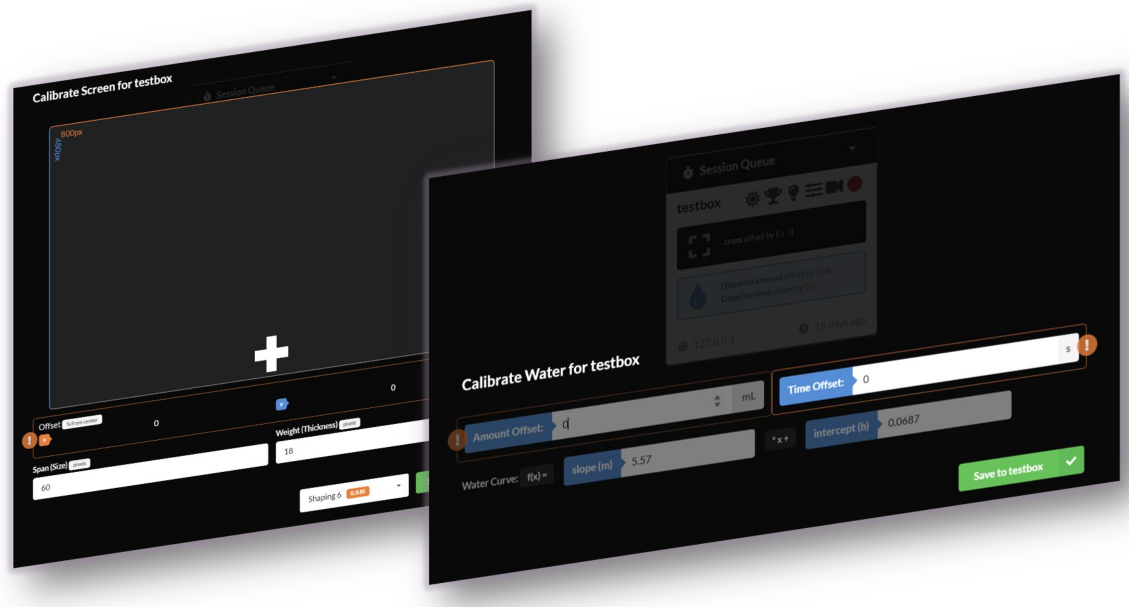

Calibrate Screen for testbox
Session Queue
800px
480px
0
Offset % from center 0
x
y
Weight (Thickness) pixels 18
Span (Size) pixels 60
Shaping 6 0, 0.85
Session Queue
testbox
cross offset by [0, 0]
Dispense amount offset by 0 mL
Dispense time offset by 0 s
18 days ago
127.0.0.1
Calibrate Water for testbox
Amount Offset: mL
Time Offset: 0 s
Water Curve: f(x) = slope (m) 5.57 * x + intercept (b) 0.0687
Save to testbox

Fig. 5. Two windows depicting a "calibration view" within the RAIL GUI. The leftmost window modal is used to align visual elements to the physical screen on each box or device, and the rightmost window modal is used to offset the amount of water reward delivered to a particular box, or the duration the solenoid valve stays open, as well as to customize the linear function used to translate the amount in milliliters to a solenoid open duration in seconds.

An update of Raspbian OS from Debian Bullseye to Debian Bookworm propelled RAIL code forward in regard to futureproofing by boosting its compatibility with new hardware, such as the Raspberry Pi 5. Since different devices and OSes may be integrated into the RAIL system, this also offered an excellent opportunity to refine how cross-platform installers will be handled during RAIL setup for both RAIL boxes and the RAIL hub. The installer recommendations from the Linux Foundation's OpenJS courses introduced me to new resources that helped to streamline cross-platform installations. Our RAIL documentation was also improved to cover common troubleshooting cases, particularly concerning the new installer and initial setup.

Each RAIL operant training box includes a Raspberry Pi which runs either the browser-based RAIL application or the RAIL desktop application in kiosk mode. Simultaneously, several background services on the Raspbian operating system (OS) maintain a connection to device hardware and software. However, the Debian Bookworm update also rendered some of the RAIL box's background services obsolete and revealed a future dilemma for the RAIL workflow.

Carefully calibrated screen brightness is very important for visually based behavioral tasks and video recordings of experiment sessions are essential for aligning behavior to neural data. However, adjusting each box manually or downloading from each RAIL box one at a time quickly becomes untenable at scale. Therefore, convenient remote access to RAIL boxes is a high priority for the overall serviceability of the system.

The Python scripts governing the background services for remote modification of RAIL box properties and Raspberry Pi settings, such as screen brightness, were updated for Debian Bookworm, and brand new functionality was added in the form of custom Windows Batch scripts for bulk video downloads from RAIL boxes to the RAIL hub. Basic video and storage space management controls were added to the development version of our RAIL GUI, as well, integrated into a device management page. Alternatively, we investigated automated uploads to third-party video hosting and file storage services, such as Google Drive. Currently, these third-party services involve authentication and API inclusion that is too complex for the simple reliable functionality RAIL aims to provide, but it may prove to be a useful community feature in the future.

Lastly, RealVNC has long been the default remote access platform for Raspberry Pi's and has provided the RAIL system with a user-friendly GUI for accessing each RAIL box. However, as of the update to Debian Bookworm, Raspbian OS no longer supports RealVNC. Over the summer, RealVNC ended its free tier plan, requiring Raspberry Pi and other open source projects to find alternative solutions for convenient remote access with a GUI. Raspberry Pi is already filling the vacuum with Raspberry Pi Connect, and RAIL will likely transition to using Raspberry Pi's VNC service as a convenient remote access option in the future. Yet, the experience highlighted some of the inherent risks involved in open source projects like RAIL, where the functionality of the software depends on compatibility between many different libraries, packages, and code dependencies maintained by groups with competing interests. For now, RAIL maintains stable remote access connections to RAIL boxes through the multiple redundancies of non-commercial RealVNC, Secure Shell Protocol (SSH), and the custom scripts recently written for use on Debian Bookworm.

THE DESKTOP APPLICATION

Expanding the scope of the RAIL system from small scale experiments to high-throughput behavioral training amplifies some of the most critical development challenges we've faced to date – namely, (1) network latency, (2) compliance with organizational security requirements, and (3) providing a convenient user experience that minimizes the abundant repetitive tasks involved in running massively parallel experiments.

Both the master and development branches of RAIL are network-dependent due to industry-standard security restrictions which limit a web browser's access to local device hardware. Nearly every behavioral training paradigm requires linking physical inputs, or real-time browser interactions, with either physical hardware outputs controlled by a local device or systematic software responses controlled by a database or native management script. This necessitates communication between the browser, local hardware, and potentially, a centralized database server. Thus, we used the IoT-friendly Message Queuing Telemetry Transport (MQTT) protocol as a workaround to transport data and commands between the browser-based RAIL application and local hardware via MQTT hardware drivers written in Python. These messages were relayed through our centralized database, or RAIL "hub," which allowed for real-time monitoring and data collection during sessions, yet generated a concerning amount of network latency. Using MQTT in combination with the WebSockets protocol as an outer "shell" helped us to minimize latency surrounding real-time interactive events in the browser, yet not enough to allay research concerns.

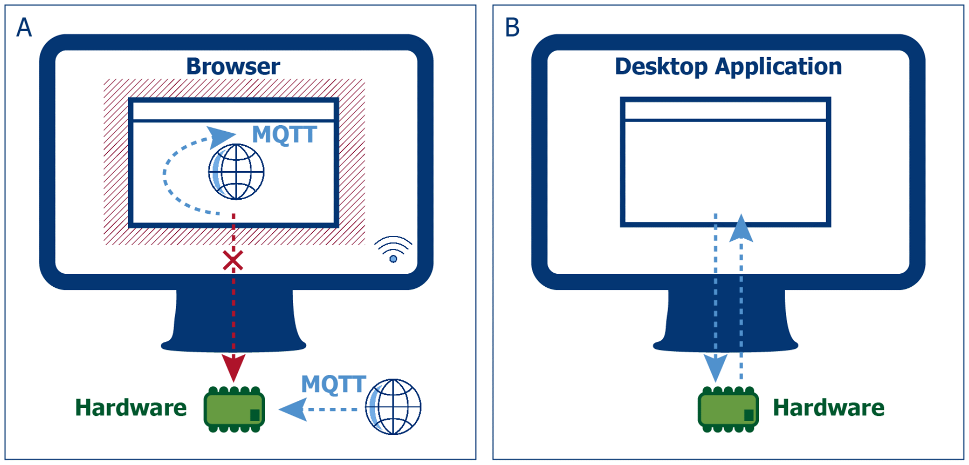


Fig. 6. Simplified comparison of two distinct networking architectures for accessing essential device hardware involved in operant training. In panel A, a network-dependent workaround utilizes the MQTT networking protocol to bypass browser limitations for accessing hardware. In panel B, a network-independent desktop application communicates directly with hardware on the Raspberry Pi device, significantly reducing the latency involved in experiment sessions.

Given many neuroscience lab's interest in integrating neuroscience hardware, such as calcium imaging or optogenetics microscopes, into their behavioral training paradigms, a more robust and reliable approach to local hardware access became necessary. Precision timing and reporting of events is crucial for the accurate interpretation of any neural data collected. Therefore, we developed a partially offline network-independent solution for communicating with our operant training box hardware – a standalone desktop application. After the initial instructions for how an experiment session should proceed are sent to participating RAIL boxes over the local network, in the form of a JSON template, the desktop application is free to execute trials without further communication with the central hub. This simulates the basic offline design of many lab-grown behavioral training systems without sacrificing the user-friendly features found in commercial software. Furthermore, it allows each RAIL box's Raspberry Pi to run the session continuously and store data locally before uploading a finalized report to the hub at the end of each session. Since trial events are no longer subject to network delays, the data-consequential effects of network latency are effectively removed from experiment sessions. A local network is only needed to synchronize data *en masse* between the RAIL boxes and the RAIL hub at the beginning or end of unqueued sessions.

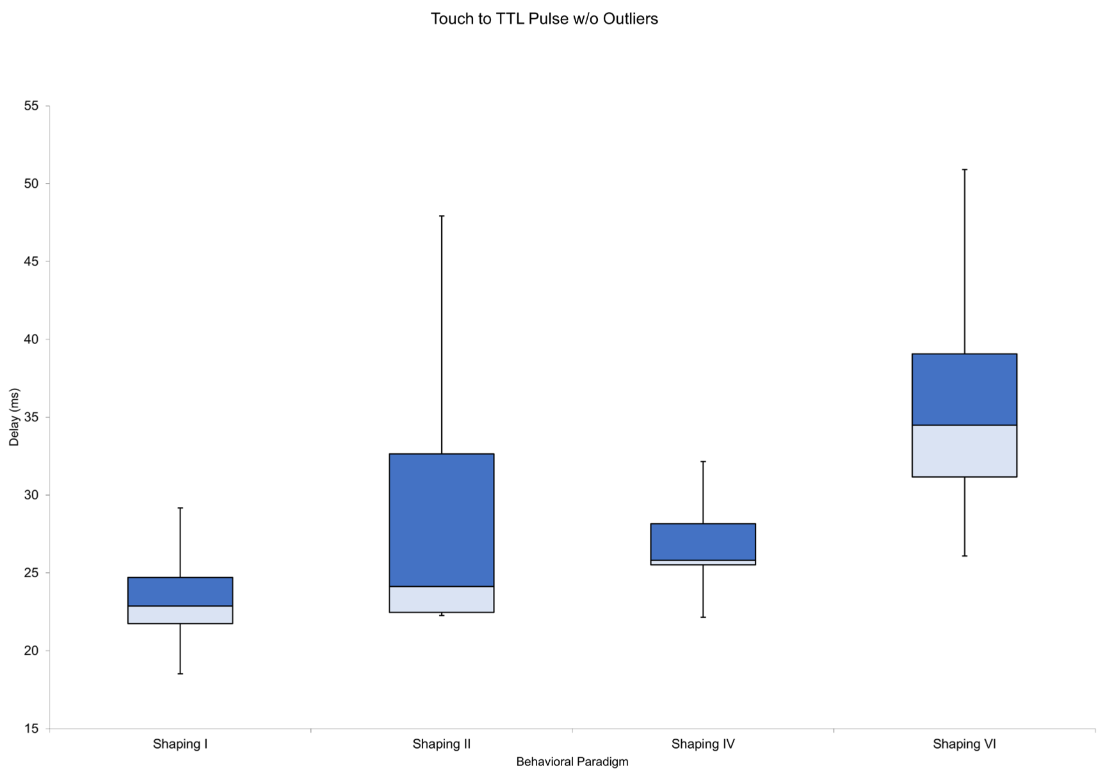


Fig. 7. Trial lag times are plotted for the first four phases of a Mysore Lab behavioral training paradigm. Each duration in milliseconds encompasses the variable network latency involved in communication between a touch input in the browser-based RAIL application and the transit of a time to live (TTL) electrical pulse from a Raspberry Pi general purpose input output (GPIO) pin to the data acquisition (DAQ) box of an Inscopix nVistaHD calcium imaging microscope.

Our planned development of a RAIL desktop application coincided with an exciting upgrade to our underlying Meteor JavaScript (JS) framework, Meteor 3.0. Years in the making, this significant improvement to the Meteor JS architecture assists in the promotion of our RAIL community building efforts. Not only does Meteor 3.0 modernize Meteor for 2024, bringing it into compliance with the greater Node.js foundation and enabling a host of new development tools, but it is presently drawing in a new crowd of software developers and fresh community support. A massive code migration is currently underway as myriad Meteor packages are being updated, some for the first time in years, to the new asynchronous Meteor format. As the updates continue, we plan on consolidating our code further, moving away from custom packages where possible and replacing package dependencies with custom code where necessary.

One of the most fundamental changes contained within the June Meteor 3.0 update was a transition away from synchronous fibers to an asynchronous "await/async" callback coding structure. As a Node-based JS web framework, Meteor is built upon Node.js but handles all of the web development tasks which are currently unsupported by Node.js. Historically, Meteor has deviated from Node.js conventions to prioritize readability and user experience, occasionally putting it at odds with Node.js and preventing updates due to code conflicts. This summer's transition to an asynchronous format finally standardized Meteor JS with the rest of Node.js, allowing Meteor to integrate Node 20, the newest version of the Node.js stack. Thus, the Node developer courses I participated in through the Linux Foundation have become even more relevant to RAIL development, especially the control flow sections on managing asynchronous operations. Since Node.js inputs and outputs will no longer be handled by RAIL fibers, it is of paramount importance that RAIL code is optimized for efficient parallel execution of events. In this regard, the lessons learned from the Linux Foundation's asynchronous lab exercises will be invaluable going forward.

Although linguistically Meteor 3.0 is a critical departure from prior implementations of Meteor JS, its release provided an opportunity to efficiently comb through and update RAIL code while performing an internal code audit. As a result, we were finally able to update RAIL to the cutting edge version of Meteor. We had previously postponed this unwieldy update due to limited time for software development, for handling deprecated packages and unmaintained software dependencies (see *Supplementary A* for a sample Meteor packages listing). Now that our RAIL code has been refreshed from this development rut, RAIL users will be able to utilize the latest web tools and infrastructure, as well as better meet modern operational security standards.

OUTCOMES & IMPACT

Many lab-grown behavioral training systems hold the potential for directed development into a more generalized community-accessible training solution. However, very rarely does neuroscience software make a successful transition from custom lab-specific software to an active open source project. When accomplished, the impacts of software like Psychtoolbox, OpenEphys, or Bpod on research capabilities and productivity are immense. Standardization across the broader neuroscience community cultivates collaboration and fosters new ideas as novel technologies are incorporated cross-contextually from outside fields. The overarching objective of the RAIL project has been to produce sustainable and impactful open source software capable of filling this need for behavioral training in neuroscience. To achieve this end, Mysore Lab needed the time and resources to focus on refining the underlying RAIL software in a multifunctional direction, rather than simply refining a few parameters or features to fit a particular study. Our participation in the FOSSProF program has given us the operational latitude we needed to expand upon our original vision for a modular and versatile behavioral training platform.

By preparing RAIL for community engagement, we are finally in a position where we can begin to build upon our network and collaborate technically with other labs. Common pitfalls of open source projects include limited support, limited functionality, compatibility issues, and the lack of a centralized provider – all issues that stem from the absence of adequate incentives for sustained development. While we suspect that addressing these concerns and cultivating a RAIL community network will be a long-term and gradual learning process, we believe that due to its relevance to a small but important niche of academic research, there are sufficient incentives already in place for RAIL development to progress in multiple directions of community support. Although most contributions will likely serve a lab-specific purpose or use case, there is already enough overlap across experimental paradigms and disciplines for any collaborative components to significantly expedite the experiment setup process for myriad labs.

In the interest of sustainability, we first discussed the marketing potential of RAIL with Andrew Wichmann from Johns Hopkins Technology Ventures (JHTV). Thanks to his guidance, we obtained a better understanding of the current business and legal landscape, as well as RAIL's position as a middle-ground option in the neuroscience software space, with combined elements of both commercial and lab-grown behavioral training systems. Going forward with this perspective, we've been able to bolster RAIL's strengths and fill in feature gaps in order for RAIL to better serve as a middle-ground behavioral training system. Modernization updates have helped with future-proofing the RAIL software and preparing for a code release to which contemporary audiences would be more willing and capable of contributing. Code review and lab-internal auditing have resulted in better security and compatibility with popular software and operating systems, and generalized features have been expanded upon with user-friendly capabilities that better showcase the system's modularity. Altogether, these improvements have prepared us for optimizing a new scalable version of RAIL and ultimately promoting a more reliable and customizable open source product to foster community engagement.

Our engagement with the wider open source community involves a two-prong approach, with emphasis on both the open source software development community and the open source neuroscience community. Although not all of our networking has resulted in meaningful collaboration, we have made significant strides by generating early interest in our future scaling-optimized and desktop-capable RAIL release.

For the neuroscience prong, we first researched other lab-grown systems that are currently being used in the successful training of mice on complex behavioral tasks. After a detailed discussion with a lab studying spatial representations in freely moving mice, we confirmed several of our suspicions on how best to address community concerns without sacrificing the userfriendliness or reliability benefits of the established RAIL system. Although labs such as this one apply many of the same components to their behavioral training systems as Mysore Lab does with RAIL, components such as Python scripts, Raspberry Pi microcontrollers, IoT hardware, or solenoid valve liquid delivery systems, there is generally no emphasis on system calibration, documentation, user convenience, or code reusability. The average experiment is mostly hard-coded, and reliability usually consists of avoiding networking and centralization. Oftentimes the system operator and the original developer of the system are far removed, with basic training passed down from researcher to researcher but no clear path forward for how to add new functionality to the system. Thus, in cases where the current researcher is forced to manage 24 different command terminal windows to start 24 different boxes running the same hard-coded paradigm, there seems to be genuine interest in an open source system that can recreate the task through a GUI and manage sessions reliably and conveniently *en masse*.

On October 7th, our new version of RAIL will be presented at the Society for Neuroscience (SfN) conference in Chicago, Illinois. This annual mass gathering of the neuroscience community attracts nearly half a million neuroscientists from labs around the globe. At one of our past presentations of RAIL at SfN, we spoke with around 14 different labs who were actively interested in the then-untested RAIL prototype and added them to our mailing list. Now that the RAIL project is at a more mature state, we expect to grow our list with additional community engagement. And this time, we will have more community building tools at our disposal to make the most of their engagement.

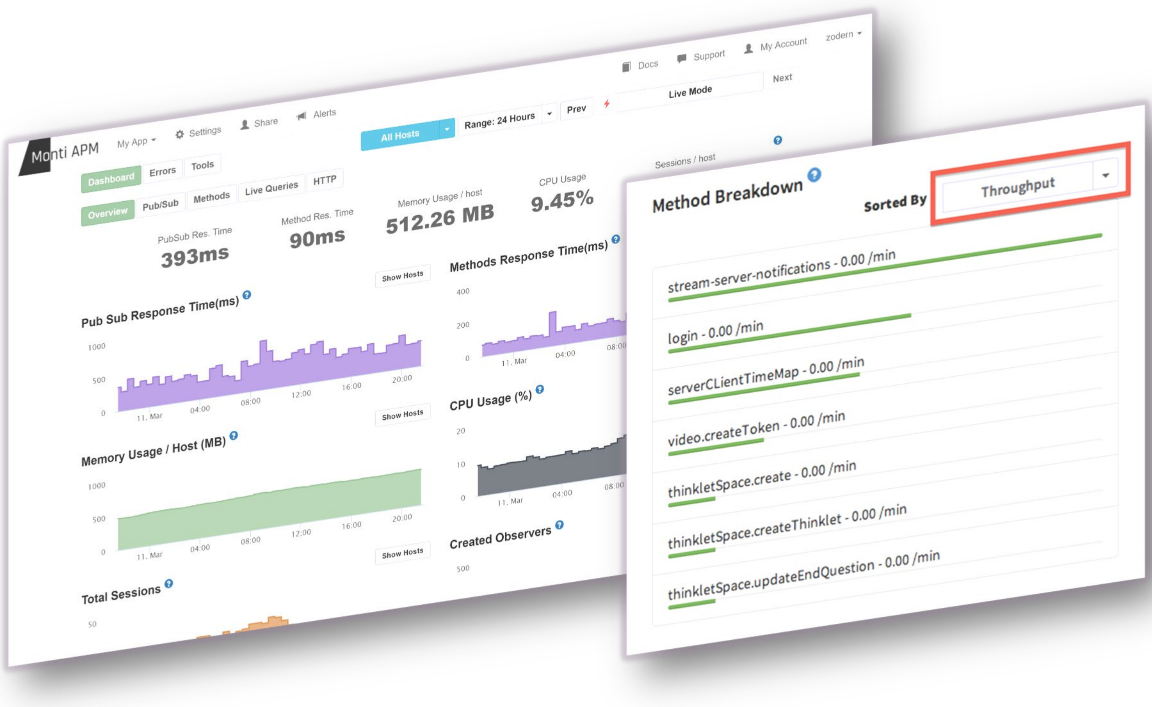
Fig. 8. The JavaScriptLandia badge conferred upon purchase of an individual membership. It appears the networking aspects of membership are only available through public connections with personal social media accounts.

For the software prong, the most promising avenues forward seem to be through advertising RAIL, our particular application or use case of their respective softwares, through the Meteor JS, MongoDB, and ScaleGrid networks. Inclusion on website showcases and participation in community forums will inspire curiosity and interest in the new community-prepared version of RAIL. Meteor Software in particular has already provided us with educational resources to share with both experienced and aspiring engineers and researchers who express interest in learning how to use Meteor JS and RAIL. Along with RAIL-specific information and resources, these materials will be distributed at SfN in October.

The Linux Foundation’s JavaScriptLandia membership was also an excellent introduction into the Node.js ecosystem via a reputable industry giant and credentialing authority. These credentials will be included in our RAIL community outreach materials.

However, in isolation, the program's networking potential appears to be limited without further purchases or interactions through third-party social media.

Future RAIL development will focus on thoroughly testing the RAIL desktop application in an experimental lab setting using funding from an R21 and other grants. While the RAIL system was actively engaged in data collection for an ongoing experiment, it was not practical to make significant changes to the working RAIL hub software. Now, after the completion of SfN on October 9th, the current RAIL master and development branches will be deprecated to make way for the desktop-capable Meteor 3.0 version. As soon as the purchasing process resolves for Meteor JS and ScaleGrid, this newest version of RAIL will be hosted on Meteor Galaxy and characterized using the Monti Application Performance Monitor (APM). In addition to monitoring system metrics like memory usage, CPU usage, and active network connections, Monti APM will allow us to drill down and optimize RAIL code at a granular methods level. Any and all performance improvements will then be translated into and expressed as added parallel training capabilities using per-box data equivalency units as described in Figure 1.



stream-server-notifications - 0.00 /min
login - 0.00 /min
serverCLientTimeMap - 0.00 /min
video.createToken - 0.00 /min
thinkletSpace.create - 0.00 /min
thinkletSpace.createThinklet - 0.00 /min
thinkletSpace.updateEndQuestion - 0.00 /min

Fig. 9. Screenshots of an example application being monitored and analyzed by Monti APM. On the left, Monti APM monitors overall system resources, data which will be used to stress test the RAIL system and calculate the total number of RAIL boxes that may be run in parallel. On the right, a method level breakdown of application usage sorts from highest to lowest throughput, allowing programmers to optimize the highest priority methods first for the greatest upfront resource savings and the quickest performance improvements.
Images from https://docs.montiapm.com/

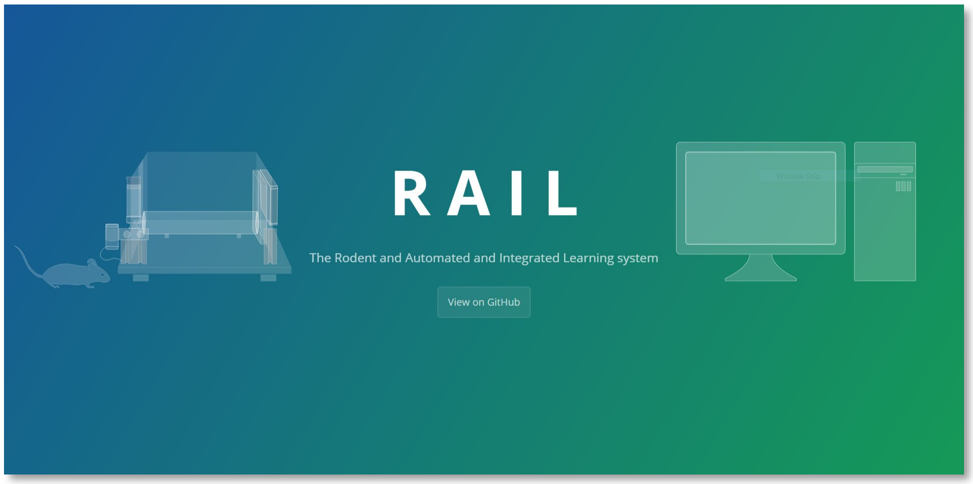
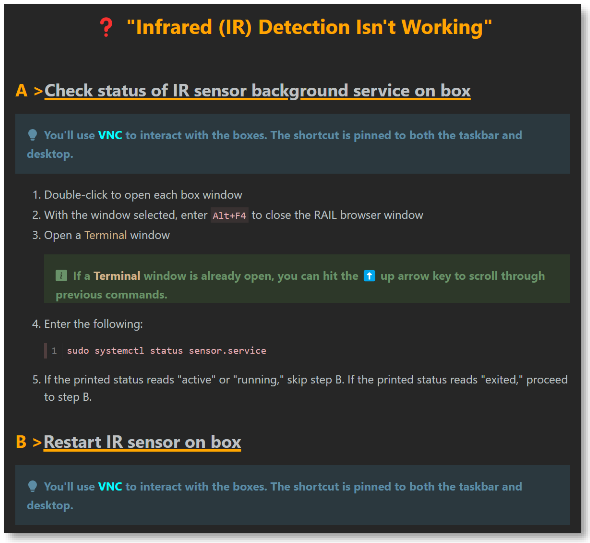


Fig. 10. Landing splash page for the RAIL project, currently hosted on GitHub Pages.

By uploading our RAIL application to Meteor Galaxy and eventually hosting our database on ScaleGrid's standalone MongoDB server, we will be engaging in vertical scaling by moving operations onto a more powerful resource-rich machine. Horizontal scaling would require we further segment our RAIL application, distributing different system capabilities across separate servers. We have already segmented RAIL services into four separate servers based on functionality – (1) a general operations Node server, (2) a real-time video streaming server, (3) a Mongo database server, and (4) a MQTT server. As we collect system performance data, we will use our premium support plans to consult with Meteor and ScaleGrid engineers on whether this is sufficient horizontal scaling for our needs or if further optimization is possible. Additionally, we will use support recommendations to optimize our database object structures and minimize document sizes, putatively leading to improved system capability and performance through a more efficient use of system resources than what was shown in our on initial measurements (see Figure 1).

Outside of scaling, documentation will be a top priority for Mysore Lab once Meteor Galaxy hosting is available. Currently, our instruction manuals, templates, Batch files, executables, and other RAIL documents are organized across three different file hosting services. All of the installers, 3D-printing schematics, and other large files that cannot be hosted on our static



❓ "Infrared (IR) Detection Isn't Working"

A > Check status of IR sensor background service on box

You'll use **VNC** to interact with the boxes. The shortcut is pinned to both the taskbar and desktop.

1. Double-click to open each box window
2. With the window selected, enter `Alt+F4` to close the RAIL browser window
3. Open a Terminal window

    If a **Terminal** window is already open, you can hit the ⬆ up arrow key to scroll through previous commands.

4. Enter the following:

```
sudo systemctl status sensor.service
```

5. If the printed status reads "active" or "running," skip step B. If the printed status reads "exited," proceed to step B.

B > Restart IR sensor on box

You'll use **VNC** to interact with the boxes. The shortcut is pinned to both the taskbar and desktop.

Fig. 11. Sample extract from a lab-internal RAIL troubleshooting manual. Here, the process for diagnosing and remedying an unresponsive infrared (IR) sensor is outlined in a stepwise fashion.

GitHub Pages webpage or in the GitHub wiki will be moved to the dedicated RAIL website hosted on Meteor Galaxy using FreeDNS. This website will help us to engage professionally with other interested labs and individuals we meet through SfN, and in general, provide a more accessible platform for any interested parties to contact us and communicate on the project.

We will also continue to generalize RAIL system properties where possible, produce more experimental paradigm and data analysis templates, and incorporate more automation and user-friendly features in support of different research paradigms. One currently planned feature set encompasses a redesign of the RAIL box for extended hours-long operation, and another entails GUI integration of the continuous data streams output by calcium imaging or optogenetics microscopes. Each new paradigm presents novel challenges to RAIL's versatility on the software side, as well as new opportunities for exploiting the modular capabilities of RAIL's intrinsic template system.

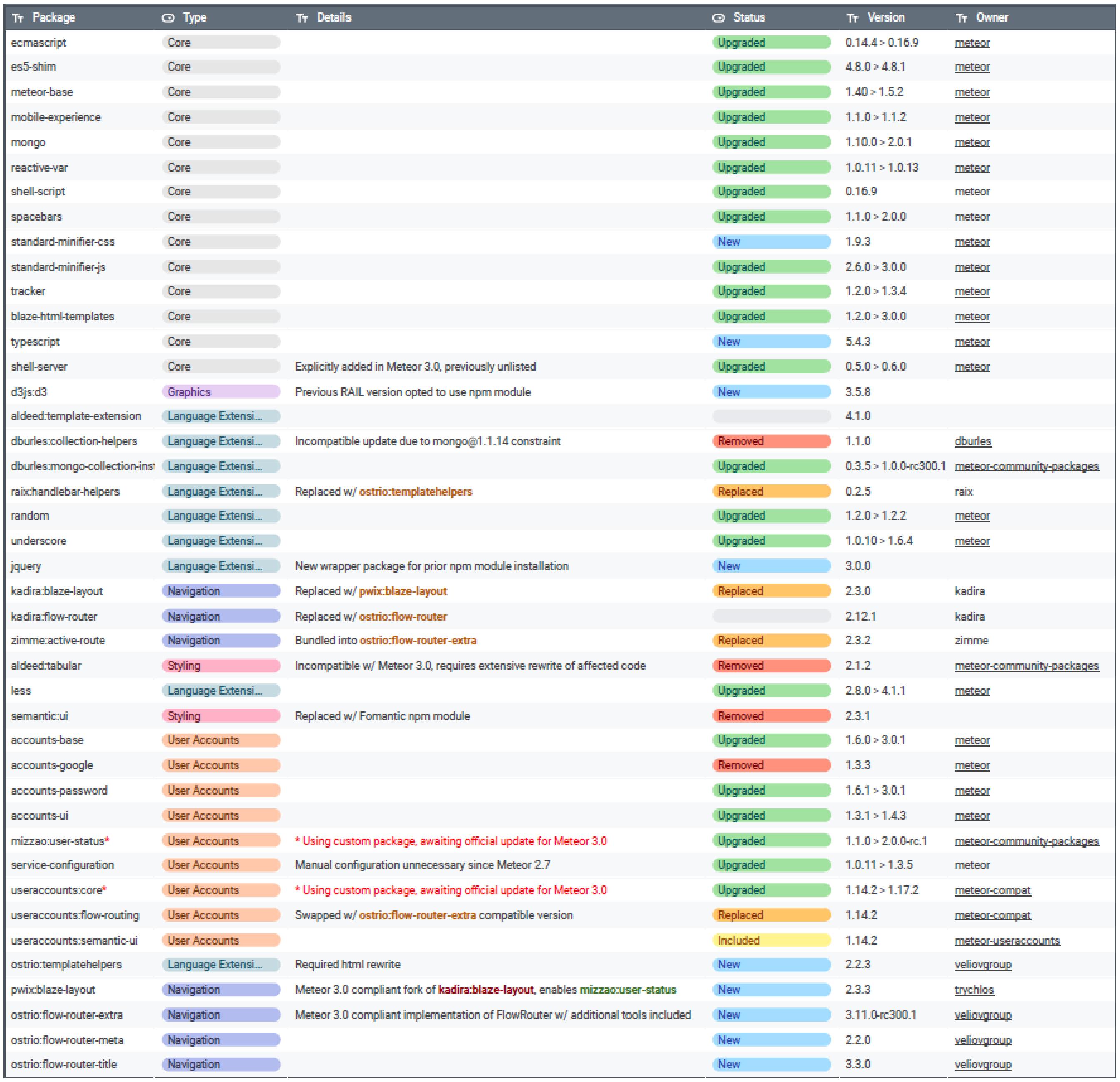
In summation, I have learned over the course of the RAIL project that open source software development is a careful balancing act of competing initiatives and tradeoffs. As there is no "one-size-fits-all" solution for most problems, developers must tailor capabilities and features towards a particular audience or use case. Oftentimes solving a problem is only the first step, as the labor involved in preparing a solution can sometimes be more intensive than the development of the solution itself.

Furthermore, in order for community participation surrounding any project to grow, there must be sufficient momentum to overcome the benefits of a "do-it-yourself" solution. In other words, a system that aims to be more generic like RAIL must also be feature-rich and convenient enough to outweigh the benefits of a perfectly tailored lab-grown solution. Reaching this critical mass inflection point for a community involves a great deal of upfront sustained effort and commitment.

Although our project's timeline did not evolve according to plan, the valuable experience of problem solving with individuals peripheral to the project, such as JHTV advisors, purchasing contacts, and database engineers, likewise resulted in unexpected advances. We gained more familiarity with colleagues and engaged in unforeseen community outreach. Ultimately, our goal is to provide an open source solution for the larger community. So, in that regard, the time delays were very well spent, learning more about other community members' unique perspectives on the issue we've set out to solve and gaining more of our own perspective, as well, on RAIL's role within the larger open source and neuroscience communities.

SUPPLEMENTARY

A



Package	Type	Details	Status	Version	Owner
ecmascript	Core		Upgraded	0.14.4 > 0.16.9	meteor
es5-shim	Core		Upgraded	4.8.0 > 4.8.1	meteor
meteor-base	Core		Upgraded	1.40 > 1.5.2	meteor
mobile-experience	Core		Upgraded	1.1.0 > 1.1.2	meteor
mongo	Core		Upgraded	1.10.0 > 2.0.1	meteor
reactive-var	Core		Upgraded	1.0.11 > 1.0.13	meteor
shell-script	Core		Upgraded	0.16.9	meteor
spacebars	Core		Upgraded	1.1.0 > 2.0.0	meteor
standard-minifier-css	Core		New	1.9.3	meteor
standard-minifier-js	Core		Upgraded	2.6.0 > 3.0.0	meteor
tracker	Core		Upgraded	1.2.0 > 1.3.4	meteor
blaze-html-templates	Core		Upgraded	1.2.0 > 3.0.0	meteor
typescript	Core		New	5.4.3	meteor
shell-server	Core	Explicitly added in Meteor 3.0, previously unlisted	Upgraded	0.5.0 > 0.6.0	meteor
d3js:d3	Graphics	Previous RAIL version opted to use npm module	New	3.5.8	
aldeed:template-extension	Language Extensi...			4.1.0	
dburles:collection-helpers	Language Extensi...	Incompatible update due to mongo@1.1.14 constraint	Removed	1.1.0	dburles
dburles:mongo-collection-ins	Language Extensi...		Upgraded	0.3.5 > 1.0.0-rc300.1	meteor-community-packages
raix:handlebar-helpers	Language Extensi...	Replaced w/ **ostrio:templatehelpers**	Replaced	0.2.5	raix
random	Language Extensi...		Upgraded	1.2.0 > 1.2.2	meteor
underscore	Language Extensi...		Upgraded	1.0.10 > 1.6.4	meteor
jquery	Language Extensi...	New wrapper package for prior npm module installation	New	3.0.0	
kadira:blaze-layout	Navigation	Replaced w/ **pwix:blaze-layout**	Replaced	2.3.0	kadira
kadira:flow-router	Navigation	Replaced w/ **ostrio:flow-router**		2.12.1	kadira
zimme:active-route	Navigation	Bundled into **ostrio:flow-router-extra**	Replaced	2.3.2	zimme
aldeed:tabular	Styling	Incompatible w/ Meteor 3.0, requires extensive rewrite of affected code	Removed	2.1.2	meteor-community-packages
less	Language Extensi...		Upgraded	2.8.0 > 4.1.1	meteor
semantic:ui	Styling	Replaced w/ Fomantic npm module	Removed	2.3.1	
accounts-base	User Accounts		Upgraded	1.6.0 > 3.0.1	meteor
accounts-google	User Accounts		Removed	1.3.3	meteor
accounts-password	User Accounts		Upgraded	1.6.1 > 3.0.1	meteor
accounts-ui	User Accounts		Upgraded	1.3.1 > 1.4.3	meteor
mizzao:user-status*	User Accounts	* Using custom package, awaiting official update for Meteor 3.0	Upgraded	1.1.0 > 2.0.0-rc.1	meteor-community-packages
service-configuration	User Accounts	Manual configuration unnecessary since Meteor 2.7	Upgraded	1.0.11 > 1.3.5	meteor
useraccounts:core*	User Accounts	* Using custom package, awaiting official update for Meteor 3.0	Upgraded	1.14.2 > 1.17.2	meteor-compat
useraccounts:flow-routing	User Accounts	Swapped w/ **ostrio:flow-router-extra** compatible version	Replaced	1.14.2	meteor-compat
useraccounts:semantic-ui	User Accounts		Included	1.14.2	meteor-useraccounts
ostrio:templatehelpers	Language Extensi...	Required html rewrite	New	2.2.3	veliovgroup
pwix:blaze-layout	Navigation	Meteor 3.0 compliant fork of **kadira:blaze-layout**, enables **mizzao:user-status**	New	2.3.3	trychlos
ostrio:flow-router-extra	Navigation	Meteor 3.0 compliant implementation of FlowRouter w/ additional tools included	New	3.11.0-rc300.1	veliovgroup
ostrio:flow-router-meta	Navigation		New	2.2.0	veliovgroup
ostrio:flow-router-title	Navigation		New	3.3.0	veliovgroup