CrossMark

Testing Modules for Experiments in Stellar Astrophysics (MESA)

William M. Wolf[1] , Josiah Schwab[2] , R. Farmer[3] , and Evan B. Bauer[4]
[1] Department of Physics and Astronomy, University of Wisconsin – Eau Claire, Eau Claire, WI 54701, USA; wolfwm@uwec.edu
[2] Department of Astronomy and Astrophysics, University of California, Santa Cruz, CA 95064, USA
[3] Max-Planck-Institut für Astrophysik, Karl-Schwarzschild-Straße 1, D-85741 Garching, Germany
[4] Center for Astrophysics | Harvard & Smithsonian, 60 Garden St., Cambridge, MA 02138, USA

Abstract

Regular, automated testing is a foundational principle of modern software development. Numerous widely used continuous integration systems exist, but they are often not suitable for the unique needs of scientific simulation software. Here we describe the testing infrastructure developed for and used by the Modules for Experiments in Stellar Astrophysics (MESA) project. This system allows the computationally demanding MESA test suite to be regularly run on a heterogeneous set of computers and aggregates and displays the testing results in a form that allows for the rapid identification and diagnosis of regressions. Regularly collecting comprehensive testing data also enables longitudinal studies of the performance of the software and the properties of the models it generates.

Unified Astronomy Thesaurus concepts: Astronomy software (1855); Open source software (1866); Stellar evolutionary models (2046); Stellar physics (1621)

1. Introduction

The stellar evolution software instrument MESA (Modules for Experiments in Stellar Astrophysics; Paxton et al. 2011, 2013, 2015, 2018, 2019; Jermyn et al. 2023) is developed by a worldwide team. The complex nature of the software itself and of the stellar evolution scenarios that it models demand frequent, automated testing in order to ensure changes do not have unintended side effects.

To this end, MESA has a collection of approximately 100 test cases spanning a wide range of stellar-modeling applications that are regularly run and checked for regressions. These test cases can take anywhere from tens of seconds to many hours to complete, varying with the particular test case and the machine running it. MESA has a large user base that runs it on a heterogeneous set of Unix-based operating systems and computers ranging from consumer laptops up to supercomputing clusters. Additionally, the global nature of MESA development requires that results from running the test suite need to be easily accessible through a cloud-based interface. These three aspects (lengthy test cases, heterogenous hardware and software, and distributed development) require a unique testing infrastructure, which we lay out in this paper.

Aspects of this testing infrastructure, including the MESA TestHub, were briefly introduced in Appendix D of Paxton et al. (2019) as a means to aggregate and digest test results on a variety of different hardware and software platforms. Since that time, the MESA testing infrastructure has been extensively overhauled to reflect changes to the MESA development workflow (i.e., a change of version control system from subversion to Git) and to expand its capabilities. Several of these updates were briefly described in Jermyn et al. (2023). This article gives a more complete summary of the current design and implementation of the MESA testing infrastructure. We hope that other research software projects with

similar needs will use our findings to support their own testing infrastructures.

2. Testing

Figure 1 summarizes the MESA testing flow. At the lowest level shown, MESA itself contains scripts that compile the software and execute tests (Section 2.1). The process of obtaining a specific version of MESA and invoking these scripts is managed by the mesa_test Ruby gem (Section 2.2), which also transmits the results of the tests. The data reported from instances of mesa_test are collected and collated by the MESA TestHub Ruby on Rails web application (Section 2.3), which then displays the results to the developers on a dynamic website.

2.1. MESA

High-level MESA[5] modules (currently star, binary, and astero) each contain a test_suite directory with a collection of test cases. MESA includes a set of shell scripts that perform bookkeeping tasks such as enumerating the available tests and providing a mapping between an integer identifier and a given test.

A shell script each_test_run is responsible for running a single test case and recording its output. When invoked, it creates a YAML[6] file testhub.yml and populates it with metadata such as the test name and the values of testing-related options. The test is executed by invoking shell scripts common to each test case directory that compile (mk), run (rn), and restart (re) the test. The run and restart scripts that invoke MESA are instrumented to record their memory usage. The output (to stdout and stderr) of these scripts is captured and in the case of failing tests, transmitted by mesa_test to a remote server for diagnostic purposes (Section 2.3.5).

Within a test case, the rn script is responsible for executing the test. A set of helper scripts abstracts the process of running

---

[5] Source code available at https://github.com/MESAHub/mesa.
[6] YAML is a human friendly data serialization standard: https://yaml.org/.
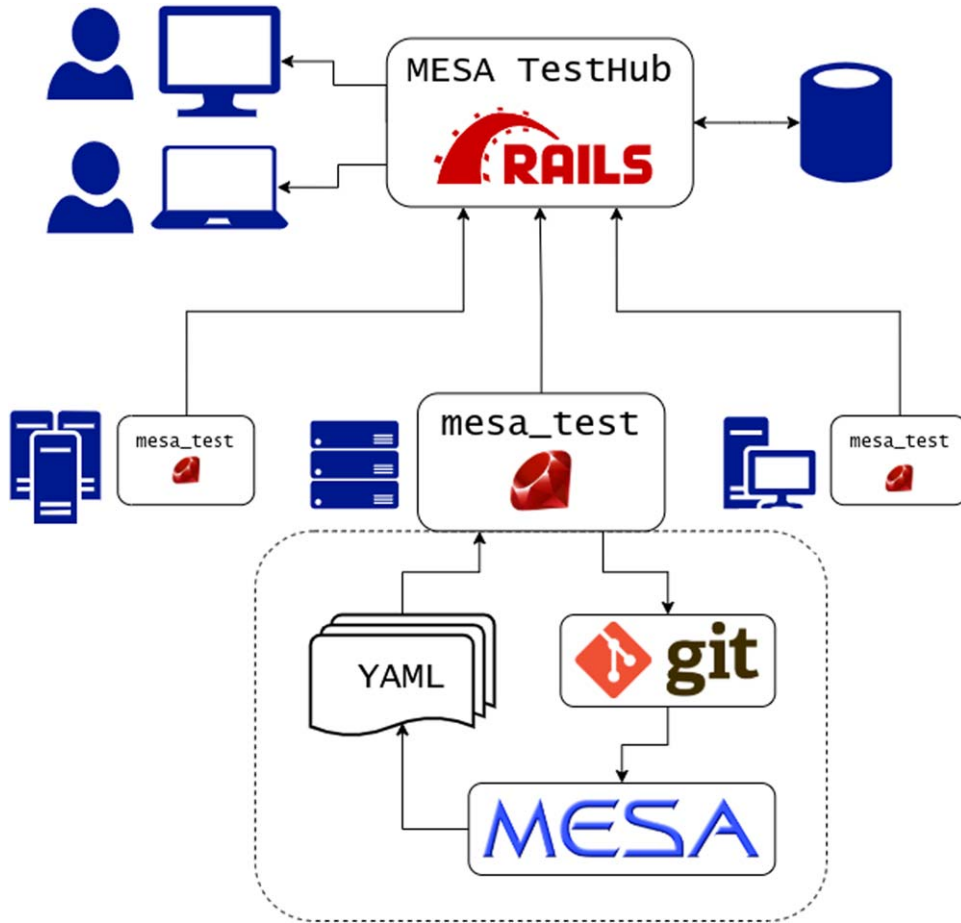
Figure 1. Workflow for testing MESA. Developers interact with the MESA TestHub via a web browser. The data for the TestHub comes from instances of mesa_test run on a heterogeneous set of computer systems (laptops, desktops, clusters). The internal operation of one instance of mesa_test is expanded and illustrated in the dotted rectangle.

a single MESA inlist[7] into a function called do_one. A rn script is effectively a sequence of do_one commands. Each call to do_one records the name of the inlist file and invokes the relevant MESA executable. When MESA exits, testing-specific code inserted into its extras_after_evolve hook appends summary information about MESA's performance such as the runtime, number of steps, number of failed solver calls, and a SHA-256 hash generated from the final model to the YAML file. Individual test cases may optionally report physical quantities to the YAML file for longitudinal tracking. In this way, when the run is concluded, a YAML file summarizing the complete run has been generated (see Figure 2).

What physical quantities to track is an area of ongoing work. Scalar quantities such as the final star mass, final core mass, or the maximum value of some quantity are trivial to track. However, more complex quantities such as the time variation in the star's mass or the radial distribution of a quantity inside a stellar model cannot currently be tracked, though we have had instances where this information would have caught a bug in MESA. In principle, the tests themselves could check time-varying quantities or radial distributions and report out a failure if something is awry. In practice, knowing what the "correct" configurations are and how to

identify meaningful deviations are difficult challenges. In a few instances, we have instructed tests to output a png file with plots of relevant quantities for more rapid diagnosis after a reported failure. However, such checks must still be done by hand as the files themselves cannot yet be used to automatically detect a failure, nor are they available online.

Many test cases allow for one or more of the slower do_one steps to be skipped if a particular environment variable is set. We refer to the inlists associated with these skipped steps as "optional" inlists. Skipping optional inlists is done to cut down on excessive runtimes, since a full run of all test cases on a single workstation can take over 24 hr otherwise. To allow the steps after a skipped step to still run, we keep saved models for the subsequent steps to load.

Skipping optional inlists obviously exposes us to missed regressions, so we have a cluster set up to do up to one full run (including optional inlists) on the main branch each day. We have also devised a way to request a full run on any commit (see Section 2.4). For a full run, testhub.yml reports data for all inlists and indicates that it ran optional inlists with a Boolean flag, as in the top portion of Figure 2.

2.2. mesa_test

A Ruby gem mesa_test[8] is responsible for obtaining a given version of MESA and using the built-in infrastructure

---

[7] An "inlist" is a generic name for a MESA input file which is a collection of Fortran namelists. The main inlist may direct MESA to read other inlist files.

[8] Source code available at https://github.com/MESAHub/mesa_test.

```
---
test_case: 1.3M_ms_high_Z
module: :star
omp_num_threads: 12
run_optional: true
fpe_checks: false
resolution_factor: 1
inlists:
   - inlist: inlist_zams_header
     runtime_minutes:    0.27
     model_number:             165
     star_age:     2.0865000907126538E+07
     num_retries:              0
     log_rel_run_E_err:       -10.5242887253877377
     steps:           551
     retries:            1
     redos:            0
     solver_calls_made:            552
     solver_calls_failed:             1
     solver_iterations:         1626
   - inlist: inlist_1.3M_ms_high_Z_header
     runtime_minutes:    1.22
     model_number:           219
     star_age:     4.6360127820968237E+09
     num_retries:              0
     log_rel_run_E_err:       -7.6596816053522883
     steps:           219
     retries:            0
     redos:            0
     solver_calls_made:            219
     solver_calls_failed:             0
     solver_iterations:         2092
mem_rn: 6180288
restart_photo: x150
mem_re: 6133712
success_type: :photo_checksum
checksum: d74822646b56937a9d07ca21efab1a37
outcome: :pass
```

Figure 2. Anatomy of a `testhub.yml` YAML file. Red text comes from `each_test_run`, green text from `do_one`, and blue text from MESA itself.

described in the previous subsection to compile MESA and run its tests. It then processes the generated YAML files and transmits that information to the MESA TestHub.

When `mesa_test` is first set up, it creates a Git mirror repository on the local machine. This is a bare copy of the remote repository (hosted on GitHub), meaning there is no associated working tree. When a user requests to test a specific version, the mirror is updated (via a `git fetch`) and a working copy is created (via `git worktree`). This process ensures that a fresh working tree is created each time and avoids having to repeatedly clone the multigigabyte repository.

After MESA is installed, tests can then be run. The design of `mesa_test` allows tests to be invoked individually, enabling easy parallelization via schedulers (e.g., SLURM) in cluster-computing environments. Individual MESA developers have created testing scripts tailored to the computing clusters to which they have access.

Figure 1 illustrates that `mesa_test` serves as the messenger between the utilitarian shell scripts that control testing in MESA and the storage and dissemination role of the MESA TestHub. Keeping the role of `mesa_test` narrow and its code as simple as possible has proved beneficial. Relying on testing scripts included within MESA rather than reimplementing them in `mesa_test` reduces code duplication and removes

opportunities for errors. More importantly, changes in the MESA TestHub or in MESA itself do not require any software updates to occur on the testing computers.

With this separation of responsibilities, `mesa_test` does not need to be updated often by developers or testers. A common type of desired change is tracking a new piece of data for each test case. To do so, the test cases simply write out the new quantity to the YAML file, and a simple migration to the TestHub database allows it to accept those new quantities. Meanwhile, `mesa_test` continues to pass the data from the YAML files to the TestHub, agnostic to its contents.

Algorithm 1. Pseudocode for the execution of `mesa_test` in parallel on a computing cluster

```
loop Every X minutes
    for commit in last N commits do
        if commit is untested then
            test_commit(commit)
        end if
    end for
end loop
function test_commit(commit)
        download_and_build_mesa(commit)

    if build_succesful(commit) then
        for module in star binary astero do
            for all tests in module do
            run_test(commit, module, test)
            submit_result(commit, module, test)
            end for
        end for
    end if
        submit_cleanup_job(commit)
end function
```

Algorithm 1 shows the pseudocode for how `mesa_test` is run on computing clusters, which we use to minimize the delay between committing code and seeing the results. Every X minutes (usually in the 5–10 minutes range), a cluster will start a management script via cron or scron that starts `mesa_test`. This script will then check for any new commits, since the last time it ran, and if there are new commits it will submit to the cluster's queue system a job that then runs `mesa_test` to download and build MESA. While `mesa_test` can run the individual test cases serially, to maximize performance on a cluster we must run them in parallel. This is achieved by iterating over all test cases and submitting one job to the cluster's queue per test case. This job is then responsible for running the test, submitting the results to the TestHub, and potentially cleaning up any files. When running multiple MESAs, each with around 100 test cases, we can generate significant network bandwidth and use significant storage temporarily during a test. Combined with the fact that each computing cluster usually runs a different queueing system, this means we must optimize and refine the testing scripts for each new environment. A version of these scripts optimized for the Helios cluster at the University of Amsterdam can be found at https://github.com/rjfarmer/mesa-helios-test.

While an interface that needs to continuously poll GitHub for changes introduces a delay between when it can start a job, it is much simpler to implement on a shared computing cluster instead of an interface based on receiving push notifications from GitHub (like the TestHub website does).

2.3. MESA TestHub

The `MESA` TestHub[9] is a web application built on top of Ruby on Rails that collects, displays, and disseminates the result of testing conducted by computers running `mesa_test`. Currently, it is hosted at https://testhub.mesastar.org. It uses a PostgreSQL database to store data associated with the `MESA` repository as well as information about registered testers, their computers used for testing, and their test-result submissions.

2.3.1. Syncing with GitHub

To organize results on a commit-by-commit basis, the TestHub must have knowledge of the `MESA` repository. To that end, the TestHub tracks all open branches and their respective commits. To accomplish this, the `MESA` repository on GitHub uses a webhook that makes a request to the TestHub every time a developer pushes commits to GitHub. Upon receiving this webhook request, the TestHub uses the GitHub application programming interface (API) to retrieve an updated listing of all open branches and their head commits. If any branch is missing or out of date, it uses the GitHub API to retrieve commit data from GitHub and update the TestHub's database.

2.3.2. Interface with `mesa_test`

The TestHub implements a simple API where authenticated https requests can deliver JSON payloads of test data which are then ingested into the database. The `mesa_test` gem streamlines this process by taking the various YAML output files from compilation and testing as well as computer information, converting this data to JSON, and then sending the result as a payload attached to an authenticated request to the TestHub server using the API.

2.3.3. The Web Front End

To interact with the data it collects, the TestHub provides a rich front-end experience on browsers that is largely powered by the free Bootstrap front-end framework with some additional JavaScript features provided by jQuery. This is also where registered users set up a computer in the database to receive testing data. There are four primary useful views in the web interface.

Single Commit View. The single commit view shows typical data for a commit, including its full SHA-256 hash, author, and message. It also shows a color-coded list of all test cases present in that commit. Test cases can have a variety of statuses depending on the results submitted so far. Tests can be *passing* if every instance of that test case submitted has passed and has either no checksum or matching checksums (for multiple submissions). They can also be *failing*, where every instance has indicated a failure in the test. They can be *mixed* if there is at least one passing and one failing instance.

Since most test case submissions include a checksum generated from the final model, a test can also have a *multiple checksum* status if all instances pass, but different computers produce different checksums for the final model. This would be concerning since most of `MESA` is written such that results should be bit-for-bit consistent across multiple platforms (Paxton et al. 2015). Notably, we do not compare checksums between submissions with optional inlists and those without,

since they would not be expected to match. Additionally, we do not track checksums for several test cases that use code that is not yet optimized to yield bit-for-bit consistency across multiple platforms, though bringing these pieces of the codebase to bit-for-bit consistency is a long-term goal of `MESA` development.

Finally, test cases can be *untested* if no instances have been submitted yet. Figure 3 shows a typical instance of this view.

Commits Index View. The commits index view lists commits in a particular branch, indicating their status in a color-coded list. Each entry shows basic information about the commit and a collection of badges indicating the number of tests that pass, fail, etc. The entire commit entry is color coded to reflect the status of that commit's most alarming test case's status, giving a quick sense of how that commit is faring. Figure 4 shows a typical example of this view.

In this view, one can also quickly toggle between different branches to see commits in context of their recent development. This is particularly useful when deciding if and when to merge one branch into another, since one would not want to merge in changes from another branch that are known to cause test cases to fail.

Test Case Commit View. The test case commit view shows data for a given test case on a given commit. This shows why a test case has the indicated status in the single commit view by listing all instances of that test case that have been submitted, including granular data like checksum hashes, runtime, random access memory (RAM) usage, and physical data related to that particular test case. Figure 5 shows an example of this view.

Test Case History View. The test case history view shows data for a given test case over many commits, either by showing commit-wide summary data for each commit (essentially the data that is expected to agree among different computers on a given commit) or by showing data from a particular submitting computer. This is useful for seeing when a test case first started failing or how some physical quantity varies with time (reflecting changes in the software). When looking at individual computers' data, this view reveals trends in runtime and RAM usage over many commits. Figure 6 shows an example of a computer-specific test case history view.

2.3.4. Daily E-mail

Being able to track the status of commits on demand via the web front end is invaluable, but since its inception the `MESA` TestHub has implemented a daily e-mail summarizing recent testing activity. Every commit in every branch tested in the previous 24 hr is reported using a similar color scheme to that indicated above, including a listing of all test cases that are failing, mixed, or show multiple checksums with links to each relevant single commit view and test case commit view. The daily e-mail serves as a way to keep the development team informed on the status of various branches. We have also set these e-mails to be sent to a dedicated testing channel in the development team's Slack workspace, where they often serve as the anchor for debugging discussions. Figure 7 shows an example of one such e-mail. In this case, the example e-mail shows the status of ongoing development on branches that have since been merged into `main` to incorporate major new features into recent MESA releases: time-dependent-convection (Jermyn et al. 2023) and carbon–oxygen phase separation (Bauer 2023).

---

[9] Source code available at https://github.com/MESAHub/MESATestHub.
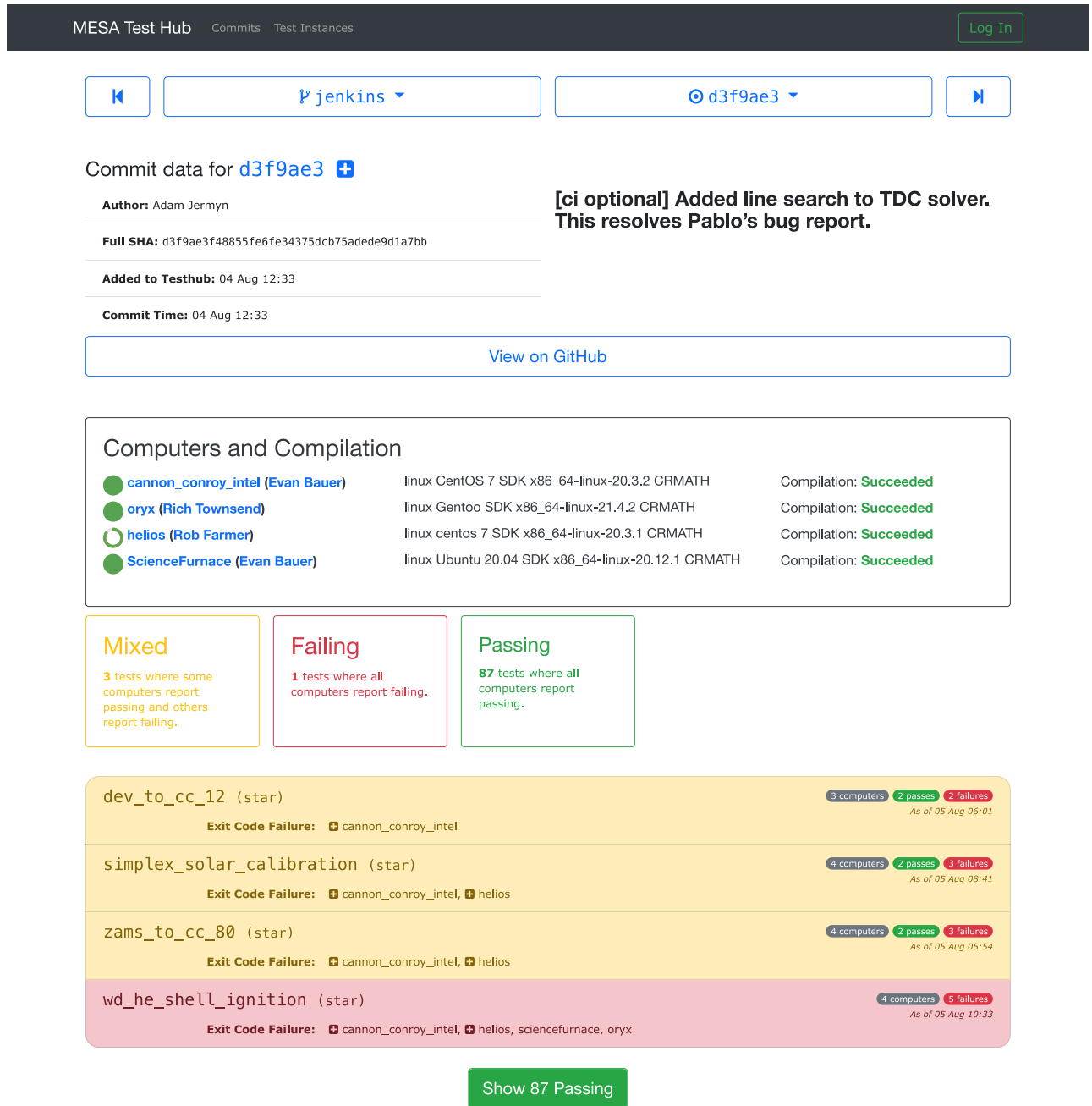
Figure 3. An example of the single commit view. Basic information about the commit itself is shown at the top. Below that is a summary of the computers that have submitted tests, where full circles indicate all tests have been run, and an incomplete circle signifies that at least some test cases have not been submitted. Below that, we see three tests with a mixed status (yellow) and one that is failing on all computers.

2.3.5. Failure Logs

Test cases inevitably fail. When they do, it is important that developers have easy access to the output associated with the failing tests. This is especially useful in the case of failures that are not easy for an individual developer to reproduce (for example, because the failure is intermittent or because it only occurs with a specific operating system or compiler). Even in the case of easily reproducible failures, this saves a developer the time needed to rerun the case and trigger the failure themselves, or to wait for the developer who owns the machine to report the failure(s) manually (especially important given the multiple time zones that MESA development occurs over).

When a compilation or test failure occurs, `mesa_test` transmits the information that was output (i.e., the contents of stdout and stderr, not the data written to the MESA history or profile files) to a separate log archiving server that is tightly integrated with the TestHub. Transmitting only the failing cases saves significant bandwidth and storage as the full logs from a complete test-suite run are $\gtrsim 10\,\mathrm{MB}$ in size. Because these files are intended for diagnosing failures, they are only retained for a limited time, under the assumption their utility is largely exhausted after issues are fixed. On the TestHub web interface, notifications of failing compilations and failing test case instances are accompanied by links to the relevant log files if they exist.
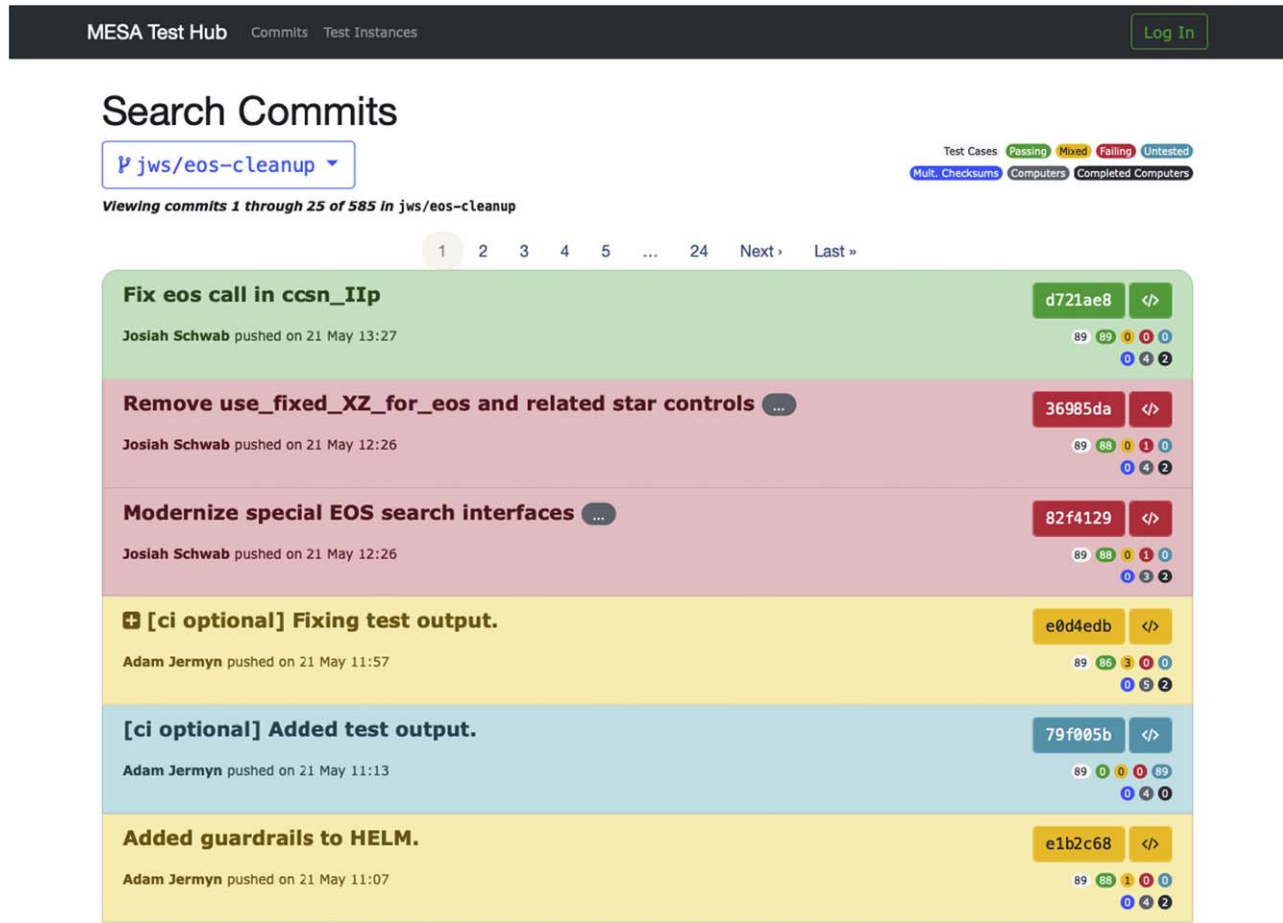
Figure 4. An example of the commits index view showing commits that are passing, failing, mixed, and untested. The "+" icon indicates that all optional inlists were run for all tests on at least one machine for that commit.

The `mesa_test` gem also includes an option to force submission of test logs even for cases where no failure occurred. We currently have one cluster set up to force test log submissions to the latest commit on the `main` branch of MESA up to once per day. This allows developers to have a quick reference to test output for passing test cases on `main` when tests report failures on other branches, while also avoiding an overload of test log files.

2.3.6. Hosting and Hardware

The TestHub is currently hosted by the Heroku[10] platform as a service on a single "basic" dyno. Dynos are isolated, virtualized Unix containers designed to work with a particular application environment. In the TestHub's case, the dyno is tuned to work well with Ruby on Rails applications. The basic level is the least capable of Heroku's persistent plans, providing 512 MB of RAM and modest access to a central processing unit (CPU) (Heroku does not publish precise details on what CPUs or how many cores a given dyno has access to) at a cost of $7 USD per month as of 2023 August. New deployments are triggered by issuing a `git push` to a special `heroku` remote, with the actual build deployment handled automatically on Heroku's end. The runtime environment can be accessed with command-line access through Heroku's website or via their command-line tool.

In addition to the web server, Heroku also hosts the PostgreSQL database the TestHub relies on via their Heroku Postgres add-on service. We use the "basic" level of service, which allows 10,000,000 rows, 10 GB of storage, and up to 20 simultaneous connections at a cost of $9 USD per month as of 2023 August. Over the first 32 months of usage since MESA development shifted to GitHub, the TestHub now uses over 2,400,000 rows spread over 13 tables, consuming 564.5 MB. So in the medium term, we may require upgraded database service, or we may opt to purge old test data to keep the data storage lean.

With the relatively low throughput the TestHub experiences, it could be served by a rather inexpensive server and a reliable Internet connection. However, the low cost of $16 USD per month and reliable service has made Heroku's platform as a service a robust and simple hosting solution. Even at the modest levels of service the TestHub runs on, it does not appear to be meaningfully bottlenecked by the processor, RAM, or database connection.

2.4. Commit Messages

To extend the flexibility in testing we have introduced a number of commit strings to pass data to the testing machines. These strings, if present in the commit message, signal a request from the committer to the testing machines to alter the standard testing setup in some way (note these are only
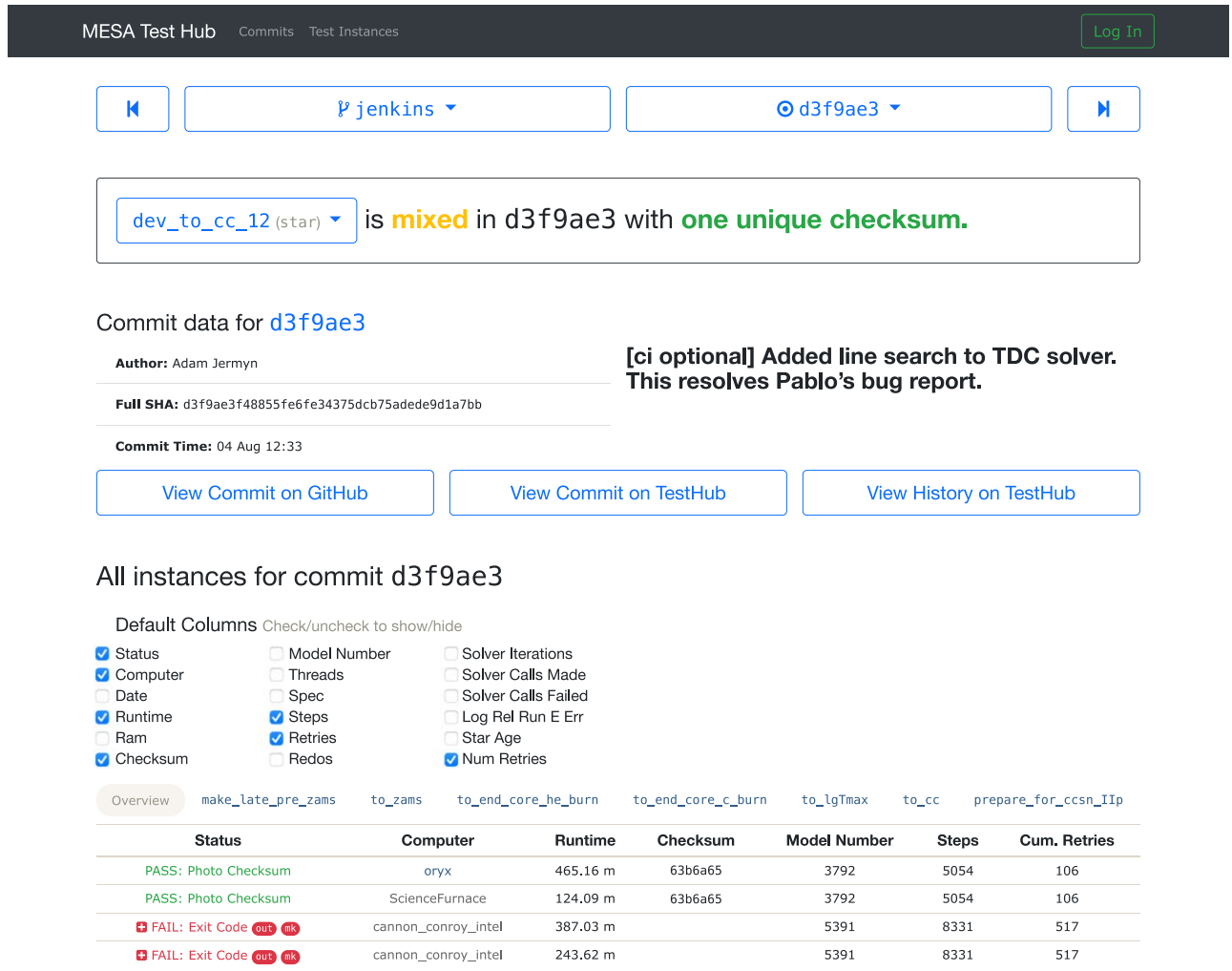
---

[10] https://heroku.com

Figure 5. An example of the test case commit view showing that the failing instances are coming from a single computer, along with links to view the logs of those instances (see Section 2.3.5). The "+" icons indicate submissions where optional inlists were run. In this example, we can see that enabling optional inlists revealed a regression in this test.

requests as not every test machine respects every commit message):

1. [ci skip] This instructs testing machines not to run the test suite on this commit. This is primarily useful when committing changes that do not affect the code, for instance documentation changes. A basic check for passing compilation is still performed and submitted.
2. [ci optional] Requests that testing machines run all optional inlists in all test cases. This is particularly useful right before attempting to merge a branch back into `main` to ensure no lurking regressions remain.
3. [ci fpe] Requests that a number of additional compile-time flags are enabled, as well as certain runtime flags that initialize data to not a number. This allows better testing for things like uninitialized variable usage and bit-for-bit issues.
4. [ci converge] Request that `MESA` runs with an increased spatial and temporal resolution, to find test cases that are numerically unconverged.

In the past, we also had a [ci split] flag that signaled that the two main clusters we were using at the time should split the tests. That is, each cluster would only run half of the tests, but between the two, all tests would be run once. This allowed for quicker turnaround during periods of rapid development. However, this required a lot of hand tuning to balance which tests went to which cluster and sacrificed the knowledge that tests worked the same on both systems (let alone that all tests could pass on a single system). Since then, we have usually had enough available computing resources that the small amount of time saved by this is not worth the complexity it introduced.

These flags enable greater flexibility in the testing. The committer can decide whether they wish for no testing ([ci skip]), normal testing (no flag), or a slower but more in-depth testing ([ci optional] and/or [ci converge]). By not having every machine respect all the flags we also ensure that the test suite is still run regularly. Indeed, we have several machines set up to test the latest commit on the main branch with all optional inlists and/or forcing the submission of logs to the log server. As a result, we will often have commits with the [ci skip] flag tested heavily if they happen to be the latest commit when this task is performed. Additional icons are also present in the web overview (see Figures 3, 4, and 5) to show at a glance which machines have tested the commit with each flag.
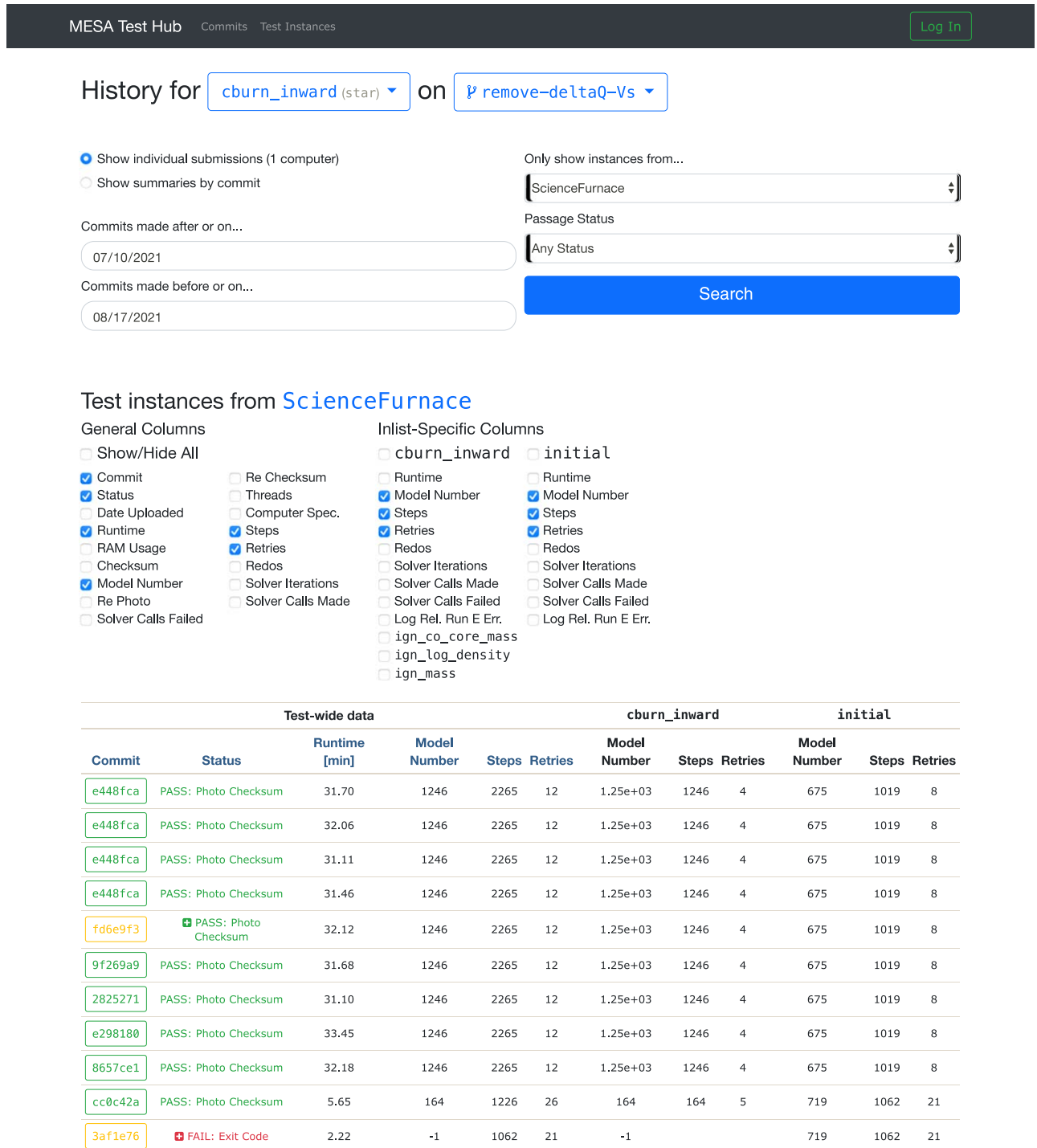
The Astrophysical Journal Supplement Series, 269:50 (10pp), 2023 December

Wolf et al.

Figure 6. An example of the test case history view showing all submissions from the computer `ScienceFurnace` for the test case `cburn_inward` over a range of dates for all commits on the `remove-deltaQ-Vs` branch.

3. Conclusions

The combination of testing infrastructure inside `MESA`, `mesa_test`, and the `MESA` TestHub has gone through many refinements as we have learned what works well and what our development team really needs from a testing system. Other continuous testing infrastructures exist, but the unique needs of the `MESA` project require a more bespoke solution. In particular, useful tests in `MESA` are long, complex, stellar evolution problems and not simple unit tests. We also value being able to report and track a heterogenous set of metrics, including test-specific physical quantities. Additionally, we need a testing framework designed to accommodate a wide variety of systems rather than being focused on a single server in the cloud, since `MESA` users work at a variety of scales on multiple platforms. And so, we designed a system that works well on individual laptops and computing clusters; we have even adapted it to work with a continuous testing server at the Flatiron Institute running Jenkins.[11]

---

[11] https://jenkins.io

main

3af1e76 (4 computers tested to date; 1 complete)

- **Author:** frank timmes
- **Commit Message:** [ci skip] documenting test case star_plus_point_mass_explicit_mdot, tweak star_plus_point_mass
- 90/90 cases tested
- cburn_inward (**Failures:** 1; **Successes:** 1)

937d508 (6 computers tested to date; 4 complete)

- **Author:** frank timmes
- **Commit Message:** [ci skip] documenting test case wd_stable_h_burn, tweak to wd_nova_burst
- 90/90 cases tested
- rsp_Type_II_Cepheid (**Failures:** 1; **Successes:** 4)
- cburn_inward (**Failures:** 1; **Successes:** 4)

TDC_on_default

c636105 (2 computers tested to date; 2 complete)

- **Author:** Adam Jermyn
- **Commit Message:** Added history column for counting the number of cells with > 0 TDC iters.
- 90/90 cases tested
- dev_to_cc_12 (2 computers)

937d508 (6 computers tested to date; 4 complete)

- **Author:** frank timmes
- **Commit Message:** [ci skip] documenting test case wd_stable_h_burn, tweak to wd_nova_burst
- 90/90 cases tested
- rsp_Type_II_Cepheid (**Failures:** 1; **Successes:** 4)
- cburn_inward (**Failures:** 1; **Successes:** 4)

WD_CO_phase_separation

e9295a9 (2 computers tested to date; 2 complete)

- **Author:** evbauer
- **Commit Message:** this branch will only test wd_cool for a while
- 1/1 cases tested

937d508 (6 computers tested to date; 4 complete)

- **Author:** frank timmes
- **Commit Message:** [ci skip] documenting test case wd_stable_h_burn, tweak to wd_nova_burst
- 90/90 cases tested
- rsp_Type_II_Cepheid (**Failures:** 1; **Successes:** 4)
- cburn_inward (**Failures:** 1; **Successes:** 4)

Figure 7. An example of an automatic daily summary e-mail developers receive from the TestHub.

We conclude with some lessons learned over the development of MESA's testing infrastructure. First, separation of responsibilities is very important. Originally, mesa_test handled downloading, installing, running of tests, collection of data, and submission of results to the TestHub. This resulted in asking testers to manually update their local version of mesa_test whenever a new piece of data was to be collected or a bug needed to be fixed. Since moving the management of individual tests and test data to MESA itself, mesa_test is much leaner and requires less frequent updating and maintenance.

Second, balancing coverage of tests with speed requires thoughtful compromise. More rigorous testing usually means a longer delay between a commit and a result, as well as a larger investment of limited CPU hours. On the other hand, sparse coverage runs the risk of missing important regressions. Our current method of using optional inlists to skip the most time-consuming portions of tests has resulted in relatively fast turnarounds while still allowing for more in-depth testing on at least a daily basis. However, we may need to make other compromises such as dividing the test cases into "probative" and "exemplary" categories for prioritized testing if the test suite grows appreciably in size without an accompanying growth in computational resources.

Third, using a platform as a service (like Heroku) has been a smooth and cost-effective experience. The portable nature of a cloud service was particularly helpful as the lead developer of the TestHub switched institutions with virtually no disturbance to the testing infrastructure.

Fourth, while automated testing has caught many bugs and issues in MESA, manually inspecting the output of the MESA test suite is still needed from time to time. Certain problems have arisen in the past that cannot be easily checked for by automated scripts. Thus a human review of diagnostic plots and model output is still needed occasionally to ensure that MESA is running correctly. Automated testing is not meant to constitute a complete replacement for the maintenance work of a team of experts covering the wide range of physics encountered in stellar evolution, but rapid turnaround of broad automated testing enabled by this platform has greatly expedited the pace of development and the rate at which issues are discovered and referred to experts for closer inspection.

Finally, we note that building and maintaining a robust testing infrastructure for a project with complex needs like that of MESA requires substantial expertise and effort at multiple levels. In the case of the TestHub and mesa_test, we needed some degree of full stack web-development knowledge, but the infrastructure within MESA required a deep understanding of the way tests can fail, how that information can be aggregated, and what information would be essential for diagnosing regressions. As a research software project grows from a single scientist to a distributed team, developers should be aware of the testing and quality assurance needs of their project and not overlook the significant investment of time and effort required to enable smooth development.

The MESA testing infrastructure has been a critical factor in enabling the continued development of this software. In the first two and a half years since the move to Git, we have averaged testing five commits per day, with over 570,000 test case instances reported to the TestHub representing nearly one million CPU hours of computation on clusters, workstations, and laptops. We hope that by documenting our testing infrastructure, other projects in computational science with similar demands might be able to build a system that supports their further development.

Software: MESA (Paxton et al. 2011, 2013, 2015, 2018, 2019; Jermyn et al. 2023, http://mesa.sourceforge.net), Ruby on Rails (https://rubyonrails.org), Bootstrap (https://getbootstrap.com), jQuery (https://jquery.com)

ORCID iDs

William M. Wolf ⓘ https://orcid.org/0000-0002-6828-0630
Josiah Schwab ⓘ https://orcid.org/0000-0002-4870-8855
R. Farmer ⓘ https://orcid.org/0000-0003-3441-7624
Evan B. Bauer ⓘ https://orcid.org/0000-0002-4791-6724

References

Bauer, E. B. 2023, ApJ, 950, 115
Jermyn, A. S., Bauer, E. B., Schwab, J., et al. 2023, ApJS, 265, 15
Paxton, B., Bildsten, L., Dotter, A., et al. 2011, ApJS, 192, 3
Paxton, B., Cantiello, M., Arras, P., et al. 2013, ApJS, 208, 4
Paxton, B., Marchant, P., Schwab, J., et al. 2015, ApJS, 220, 15
Paxton, B., Schwab, J., Bauer, E. B., et al. 2018, ApJS, 234, 34
Paxton, B., Smolec, R., Schwab, J., et al. 2019, ApJS, 243, 10