Promoting scientific software quality through transition to continuous integration and continuous delivery

Anja Schubert[1] and Robert M. Argent[1],

[1] Bureau of Meteorology, Victoria, Australia

Abstract

Software quality has been an issue for decades in many areas of scientific modelling for environment applications. Much of the software that has been developed is well-suited to supporting research investigation and application in one-off curated application environments, such as delivering solutions through community-based participatory approaches. However, when releasing the software into constrained production environments, with everyday operational challenges such as missing data, unintended user inputs, variable data quality, and values outside the bounds of those previously tested, lacks in software quality can become apparent. The lacks can result in lost time and effort to get the model running again, loss of trust in the model by users and, potentially, cessation in model use, and early and unexpected end-of-life. There have been many efforts through the years to encourage improved software quality, through approaches such as test-driven development, paired programming, more rigorous documentation, and better and broader user acceptance testing. The basis of many of these approaches is the art and science of software testing, through the whole development chain from unit testing, regression testing, integration testing, functional testing and to user testing. Continuous Integration/ Continuous Development (CI/CD)/ Continuous Deployment uses an automated or semi-automated process pipeline that promotes or progresses software through various stages of 'readiness' in different system realms, from development to testing to production. Faithful application of CI/CD promotes good quality software by requiring the software to pass a series of pre-determined tests before it is accepted into the next realm. Models must meet quality levels and pass automated tests to be able to be promoted. Additionally, governance gateways are used to check non-automated workflow components, such as manual testing. This paper explores the challenges and lessons learned in adoption and application of secure CI/CD in an operational environmental modelling enterprise, and suggests a minimum viable good practice approach for application to scientific environmental modelling.

Keywords

software quality; software testing; continuous integration; continuous development

1. Introduction and operational software context

The Australian Bureau of Meteorology (Bureau) is the operational agency delivering trusted, reliable and responsive weather, water, climate, ocean and space weather services for Australia – all day, every day. The Bureau also develops and supports scientific software for operational purposes, and is part of global partnerships that share science knowledge and software code through formal and informal channels. The combination of research and operational contexts, along with significant amounts of development via

Correspondence:
Contact R. Argent at Robert.Argent@bom.gov.au

Cite this article as:
Schubert, A., & Argent, R.M.
Promoting scientific software quality through transition to continuous integration and continuous delivery
Socio-Environmental Systems Modelling, vol. 6, 18779, 2024, doi:10.18174/sesmo.18779

Socio-Environmental Systems Modelling
An Open-Access Scholarly Journal
http://www.sesmo.org

internationally shared code repositories, creates a complicated situation when seeking assurance on software quality.

The operational environment requires software code that is robust to the variations that occur in 24/7 operations, such as missing or incomplete input files, dynamic input boundary conditions from companion models, variable timing of delivery of model output within chains of running models, and unplanned failures and failovers of computational platforms. A typical daily cycle of model runs may involve:

- assimilation of many millions of observations from multiple sources, both within Australia and overseas,
- computational running of a fleet of more than forty interdependent models representing high level dynamics of physical atmospheric, terrestrial and oceanic domains;
- additional 'on demand' modelling for severe events, such as tropical cyclones, and
- a wide range of downstream user systems, both within and outside of the organisation.

Therefore, the requirement for models and data systems to be robust, resilient, efficient and of high quality is essential, and the role of software testing in this is paramount.

The research reported here was undertaken in the context of an uplift of modelling code to operate on a new supercomputer. It relates to the opportunities for checking and improving software quality through a formal code 'test and build' framework when migrating from a secure manual code change system to a system built around secure continuous integration, continuous delivery and continuous deployment technologies.

Many of the models developed and used in this operational environment have the features of scientific software (Heaton & Carver, 2015; Kanewala & Bieman, 2014), which can include variable code construction and code quality, different levels and systems of version control, inconsistent test coverage and test quality, different amounts and standards of documentation of the software and its quality, and lack of robustness and portability across different computing platforms and environments. These features are often mitigated in the global operational environmental modelling community, where professional practices of shared repositories, code review and approval, specified testing requirements and formal release and deployment mechanisms are in place.

Moving from the research to the operational realm within a given organisation can involve changes in the workplace setting. Generally, there is greater freedom to explore, investigate and share in the research computing realms, and gradually less freedom and greater security controls as code passes from research to development, to test and, finally, to production environments.

This paper explores the practicalities of testing and improving code quality in a research-to-operations environment, in the context of uplift of an entire code ecosystem in migrating to a new computing platform and technology stack. We explore our experiences through three case studies and describe the impacts of different levels and natures of test coverage on the transition of models to production status in the new environment. Reflecting on the problem situation, we suggest a minimum viable good practice approach for application to scientific environmental modelling.

2. Literature review

The concepts and practices of formal code testing have been part of professional software engineering for decades. Practices include model-based testing (e.g. Dalal et al., 1999), model-driven architecture based testing (e.g. Uzun & Tekinerdogan, 2018), test-driven development (Beck, 2002), requirements engineering through conceptual models (Gupta et al., 2022), metamorphic testing (Kanewala & Chen, 2019), and, at a meta-level abstracted from the actual code, requirements testing (dos Santos et al., 2020). In recent times these practices have become of more interest in the realms of research development of scientific software (e.g. Iwanaga et al., 2018). The scientific and operational context of the Bureau of Meteorology sits across both these camps. Various systematic literature reviews over the past decade provide good grounding for this work and Jakeman et al. (2024) provide a broader context for the role of software quality in environmental modelling practice.

Kanewala & Bieman (2014) offer an early comprehensive survey of the state of testing of scientific software. They identified two primary groups of challenges – cultural, due to the nature of scientists and scientific endeavour, and technical, due to the nature of the software. One of the technical challenges that is a factor for researchers is clear identification of an 'oracle' in testing – in complex environmental modelling there is often not a single 'correct' answer, but rather a set of outputs over time that accurately represent the behaviour of the system under consideration, within tolerances. Kanewala & Bieman (2014) offer approaches to overcoming oracle problems, including reference data sets, professional judgement, simplified data sets and using experimental or model outputs, rather than observations. One factor not mentioned is the cooperative use of shared code repositories across multiple organisations, and modification that might therefore be required to the nature of testing, to give assurance on software quality for all users.

A systematic literature review focused on good software practice used in science domains (Heaton & Carver, 2015) found that although considered important, testing, verification and validation were neither widely nor consistently adopted. Primary drivers for this were considered to include the complexity of the applications and the different range of approaches that might be required to be adopted to reach a significant level of coverage and assurance.

The investigation of scientific software quality was carried further (Nanthaamornphong & Carver, 2017) with a survey on the use of Test Driven Development (TDD). In TDD (Beck, 2002), developers focus first on specifying and writing tests to ensure software meets the specific requirements of users, then write code to pass the tests. TDD is probably easiest to implement when starting a new system, rather than being retrofitted to code that has been modified and refactored many times by many different scientists over generations of versions. At a high level, they found that good TDD, properly implemented, improved overall software quality with fewer defects. However, the timing and nature of effort required was a possible challenge for some situations. This survey provided good context for the problem situation represented here due to the different nature and coverage of the tests used. Of interest in the context of shared code repositories is the role of refactoring identified by Nanthaamornphong & Carver (2017) as critical to the TDD process. The scope for refactoring may be considerably constrained for us due to the partnership approach to code and the need to possibly bring a group of disparate developers together to understand and accept the refactoring. This may be particularly challenging if developers have not worked on the code for a number of years and have forgotten the details, although still hold a strong sense of ownership of the code and are sensitive to changes.

In parallel with the growth of interest in improving software quality by testing amongst developers of scientific software systems has come development of tooling and practices for software management that assist or require improved software quality and testing practice. An overall direction has been towards more automated testing, to a level where even game developers are advancing automated playtesting (Pereira et al., 2022). Automation comes to the fore with the use of the so-called continuous practices. Shahin et al. (2017) presented a systematic review of the state of practice with these, covering Continuous Integration (CI), Continuous Delivery (CD) and Continuous Deployment (CDo). The importance of these continuous practices was increasingly recognised in commercial industry and they are core to the study presented in this paper. Although Shahin et al. (2017) gave no apparent explicit consideration to development of scientific software and the challenges therein, they did recognise that "adopting continuous practices is not a trivial task since organisational processes, practices, and tool (sic) may not be ready to support the highly complex and challenging nature of these practices". For our purposes we focus on CI/CD wherein code is integrated into the main trunk/repository frequently, and tested and built and made ready for deployment, including passing pre-existing regression and integration tests and quality checks. In scientific software and complex operational environments the next step, being automated deployment, requires a level of automated testing, quality checking, release approval, deployment approval and actual deployment that is beyond the scope of the work presented here. Overall, Shahin et al. (2017) found that there were many approaches, practices and toolsets in play, and that a variety of critical factors (whether in standard industry settings or research and scientific settings) determine success in adoption of CI/CD, such as testing effort and testing time, team culture, good design principles, the application domain, and infrastructure that is appropriate to the problem situation.

An approach with potential merit to tackle the challenges of effective and broad testing of scientific software is the metamorphic testing approach (Kanewala & Chen, 2019). Although the examples presented were simplistic, the basic testing approach, to use metamorphic or change relationships to express a test in the form of expected changes to outputs arising from known changes to input (within defined tolerances) has promise. This approach

can lend itself moderately well to unit testing, to give an approach that is somewhat flexible, but still quantitative, to the nature of the unit and the inputs. In this way, metamorphic testing was suggested by Kanewala & Chen (2019) as a way to identify a broader range of known good outputs (KGOs) that would deliver a pass for a test than may have otherwise been the case.

Increased adoption and broader use of formalised testing was found (Arvanitou et al., 2021) to have occurred in the scientific software arena, albeit with only limited focus on the need for models to run operationally. Some of the practices identified as contributing to overall increases in software quality included code reuse, use of third-party libraries, and the application of ‘‘good’’ programming techniques. For testing, Arvanitou et al. 2021 found that the practices mentioned previously, such as test-driven development and use of regression testing, had increased in importance in the practices of developers.

A logical expectation of the increasing rigour around testing and software quality for scientific software would be to start to see individual case studies, particularly of models that were well-known in specific domains. One recent example of this (Peng et al., 2021) explores the testing environment for the widely used Storm Water Management Model (SWMM). The authors explored many of the considerations that occurred in our research, especially the challenge of identifying an oracle or 'true and correct' model output, against which model results arising from changes could be compared. They also emphasised the importance of having sufficient test coverage and distinguishing the purposes and meaning of the results of both unit and regression tests.

A systematic review of literature on TDD (Staegemann et al., 2023) produced a relevant and timely set of guidelines for applying TDD. In the context of our problem situation, four of these were identified as being of high relevance, particularly in a problem situation where testing is increased after initial code development, or during code uplift:

- #4. The process of creating the tests also helps in accurately determining the application’s requirements. Consequently, tests are not just tests but also become part of the system’s specification.
- #11: The development of tests should be directed at parts that might actually break.
- #12: Test automation allows for frequent regression testing with limited effort and should therefore be used a lot. However, manual testing still remains an important complement.
- #14: Issues identified during regression testing have to be fixed immediately.

The literature shows that there are many challenges to effective and robust testing in scientific software engineering, and that these challenges can be addressed in full or in part by combinations of cultural and technical practices. The following sections present our experience of these in moving software from a less constrained to a more constrained quality environment, where the nature of the infrastructure demanded high levels of effective testing to meet the everyday requirements of running software within an operational environment.

3. The operating environment

The operating environment within which our code is passed from development (DEV) to production (PROD) status is not overly complex. It is more formal and controlled than in many research environments, where, for example all development can be done in one environment before the code is packaged and deployed to a user. Coding entering the environment at the 'Development' end of the chain can generally come from one of three sources:

- External sources of standard tools and libraries, such as those used to perform a wide variety of standardised procedures, including time zone and time difference management, spatial data standardisation and grid manipulation, and extract-transform-load actions.
- Science code repositories, including those of formal research partners, with controlled code sources and defined and managed practices for extracting code from defined branches or the main code trunk, and also for submitting new science and adding bug fixes.
- Our own research environments, through code sources within the research realm, including both high performance computing such as Australia's National Computational Infrastructure (NCI) and local individual code bases.

All code is version controlled by a singular gitlab instance, with code packaged via a centralised package manager regardless of the original source. Internal and external packages are available via the same mechanism.

Figure 1 shows our research to operations environment consisting of four realms:

- Development (DEV), where the research-level code is transitioned into the start of the formal pipeline. Activities in this realm include unit and regression test development, expansion or confirmation (depending on the test coverage prior to entry to the realm), feature or enhancement development, some level of refactoring (depending on source and code quality), any optimisations and adjustments to get the code to run under control of the scheduler in the high performance computing (HPC) environment, and any hindcasts or case studies needed to assure that the model is producing the expected results. To exit the DEV realm, documentary evidence is required of test coverage and software quality assurance as well as configuration in the CI/CD pipeline.
- Service (SERVICE), the first realm accessed exclusively via the CI/CD pipeline. This exists to test the deployment of the software. The exit requirement for this realm is deployment without error and at least one successful cycle of the application.
- Test (TEST) where the majority of testing is done, particularly with downstream systems and users. This includes User Acceptance Testing (UAT) and extended system trials, to provide assurance that code runs stably, operates appropriately within the environment, such as with CPU usage, cleanup and memory use and management, and delivers the expected output to the expected location within the prescribed timeframe. The exit requirements for models to leave the TEST realm include documented evidence of completed testing, including UAT, with an acceptable backlog of defects or, at best, no defects, as well as complete 'Go Live Documentation' (GoLD) covering approvals, service levels, and support and response actions.
- Production (PROD), where the models run on a defined schedule and deliver quality assured output to users as an input to decision making. PROD systems have significant monitoring to ensure operations are performing to defined service levels, as well as to identify and resolve or escalate issues.

Note that, in line with good practice, code only passes one way through the system, with any code changes arising from issues, incidents or bugs, being tested and fixed in the DEV realm, before progressing back to PROD through SERVICE and TEST.

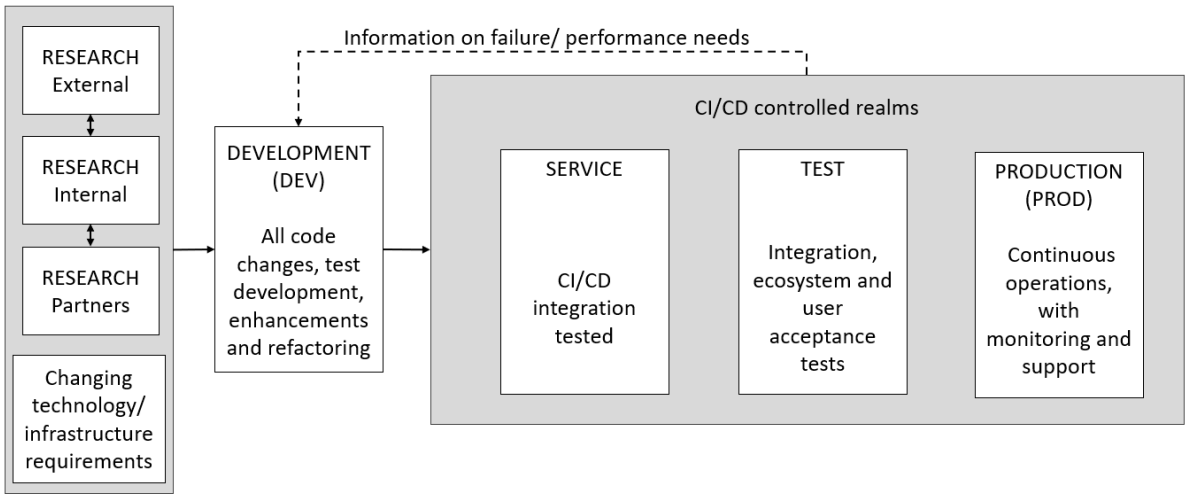

Figure 1: Realms of the Operating Environment

3.1 Testing definitions

Secure and architecturally approved design is used at the highest level to assure the quality of the software, while UAT is used to validate that the model outputs meet requirements and are fit for the needs of users. In addition to these overarching controls, four types of scientific testing are used in the DEV and TEST realms –

noting that unit testing and other basic software engineering quality functions are generally performed during initial code development, so largely occur upstream of the DEV realm.

Aligning with many features of standard software quality practices (Dubois, 2012; Kanewala & Bieman, 2014; Staegemann et al., 2023) we have four specific testing foci. These are regression, hindcast, integration, and ecosystem tests.

- **Regression testing** assures that non-scientific code changes do not affect function. These are generally performed by checking that code outputs match (within defined tolerances) exemplars or oracles (such as Known Good Outputs, KGOs) when the model is loaded with known good inputs (KGIs). When migrating code to a new supercomputer the output of the current operational system is considered the best KGO.
- **Hindcast and case study testing** extends regression tests for models with higher complexity cycling through multiple time-series input periods. These can include situations when model cycling (say, tens to hundreds of cycles) can introduce instabilities or cascading errors that only become apparent after sufficient cycles. Extended cycling is also used where the KGOs are verified statistically over hundreds to hundreds of thousands, or millions, of output data points.
- **Integrated testing** runs in real time, where the model is moved from pre-canned KGIs to real-time inputs for operational systems. In environmental applications this can include assimilation of millions of environmental observations for each run. This situational change introduces performance dependencies for reading and writing data and includes tests for performance issues such as with compute load and memory management.
- **Ecosystem testing** covers the full set of models and data management systems that need to work together. One of the challenges of complicated operational environments where there are multiple interacting dependencies between models and external data sources, such as satellite feeds and external models, is to ensure that not only do single models perform as required, but that the whole model ecosystem delivers outputs to users and downstream systems in timely and correct ways.

3.2 Continuous practices

Standard software engineering approaches, including continuous practices, are used through the above environments, and include code version control, repository management, CI/CD and code delivery.

- Code version control is used as standard practice – all code on the HPC system described above is under version control. This also includes code for any common libraries or tooling that is developed, managed and maintained within the environment.
- All binary resources that are generated within the environment are centrally managed by a formal repository manager system for both development and production uses.
- CI/CD tooling supports continuous development and continuous delivery of code. This occurs through sequences of tasks (a 'job' that is, generally, run in a container) that are packaged to form a pipeline for execution.
- Finally, a delivery service runs as part of the continuous delivery. This service delivers code as it is automatically built, and can continuously scan, test, package and deploy the code to the target realm. This capability assists developers to regularly assess the quality and stability of a code repository under change, and contributes to code quality through a repeatable, auditable and, especially, non-manual process.

The combination of automation, technologies and testing requirements provides an environment that requires good code quality and contributes to an overall working approach that values testing and supports good practice.

4. Investigation

Three case studies of model migration to the new computing environment are provided, each representing a different level of software complexity and partnership development and each of which, therefore, represents a different problem situation for code migration. Each case will describe the functions and native test coverage of the model or model suite as it enters the DEV environment, as well as the testing undertaken to assure a level of software quality that allowed the model suite to progress through the realms from DEV to PROD.

To benchmark the code pre-, and post-migration, a software maturity assessment was performed at the beginning and end of the uplift. The maturity assessment calculates a Software Maturity Index (SMI) based on six aspects:

1. Robust for operations.
2. Easy to validate.
3. Performance expectations met.
4. Easy to understand.
5. Easy to maintain and extend.
6. Has scientific integrity.

These aspects represent the fundamentals for good operational code that can be readily supported by different people through the life cycle of the model. They represent a balance between the quality and performance of the code and the needs for ongoing or occasional maintenance and enhancement. The aspects are weighted equally for simplicity.

Each aspect (1-6) is assessed against a list of expectations (Table 1). Assessment of software quality is not objective, although the assessment for levels 2, 3 and 4 is based upon a list of requirements for each level, for each model.

Table 1: Software maturity levels and expectations.

Rating level	Expectation
-1	No expectations are met. Software does not meet the basic standards of the organisation for an operational system.
1	System must meet the basic standards of the organisation for an operational system. Examples include: • Code under version control in an enterprise system; • Code is compiled and built automatically, to an extent that is repeatable; • A user guide or similar documentation is available; • Verification results are documented and are reproducible; and • A business owner, subject-matter expert and maintenance owner are identified, by name or role.
2	System should meet the requirements of Rating 1 and the model-specific criteria for a rating of 2. Examples include: • Performance standards are clearly documented; • Verification procedures are clearly documented; • Verification data are available for at least one case; • Developer documentation is available; and • Software maturity plan is complete.
3	System should meet the requirements of Rating 2 and the model-specific criteria for a rating of 3. Examples include: • Business logic is clearly separated from procedural code; • Issues can be diagnosed quickly and easily, with metadata available for each model task; • Code is modular; and • As much code as practicable is in a higher-level language (Python by default).
4	The system should not need further improvements to its software quality. There may be some beneficial work to be done, but those benefits represent 'nice to have' aspects of the system which will not substantially change the current or future costs of running, maintaining or developing the system.

The SMI is calculated as the average rating across all aspects. The highest score for the software maturity index is 4, and the lowest is -1. If a system scores less than 1 it is deemed not suitable for a production environment. It is possible for a system to have a score greater than 1, and still be assessed as not suitable for production, such as if it scored -1 on the scientific integrity aspect. For us, the SMI is primarily used as a tool to identify applications that need replacement or uplift, with the individual aspects demonstrating where work is needed

the most. We have additional requirements beyond the SMI for areas such as scientific quality, operational robustness and performance. These are governed by peer reviews and "Gateway" considerations, mentioned in Section 5.

In addition to assessing SMI, key developers involved in the migration of the three test cases were interviewed on the following:

- The impact that any existing tests had on the migration activity;
- Level of alignment of existing tests with the CI/CD pipeline in the new production environment;
- The effect that increased test coverage, or lack thereof, had on developer wellbeing;
- Any changes the developer would have made in the migration activities or the software application with regards to testing; and
- Any other general insights the developer had on migration.

4.1 Aggregate Sea Level Model

The Aggregate Sea Level model is a data aggregator that brings together data from a range of sources to produce estimates of tide levels at over 100 locations around Australia (Taylor & Brassington, 2017). These levels are more accurate than astronomical tide levels as they account for observed sea levels, forecast atmospheric pressure and other dynamics. Aggregate Sea Level, first delivered to operations in 2016, is amongst the lesser complicated modelling suites used in our forecast value chain. Output quality of the operational model is assured by statistical comparison of model output against observations.

In simplified form, the workflow to aggregate various data to produce the sea level forecasts (Figure 2) consists of:

- Data ingest and standardisation/regridding;
- Simple data transformations;
- Bias correction; and
- Linear superposition.

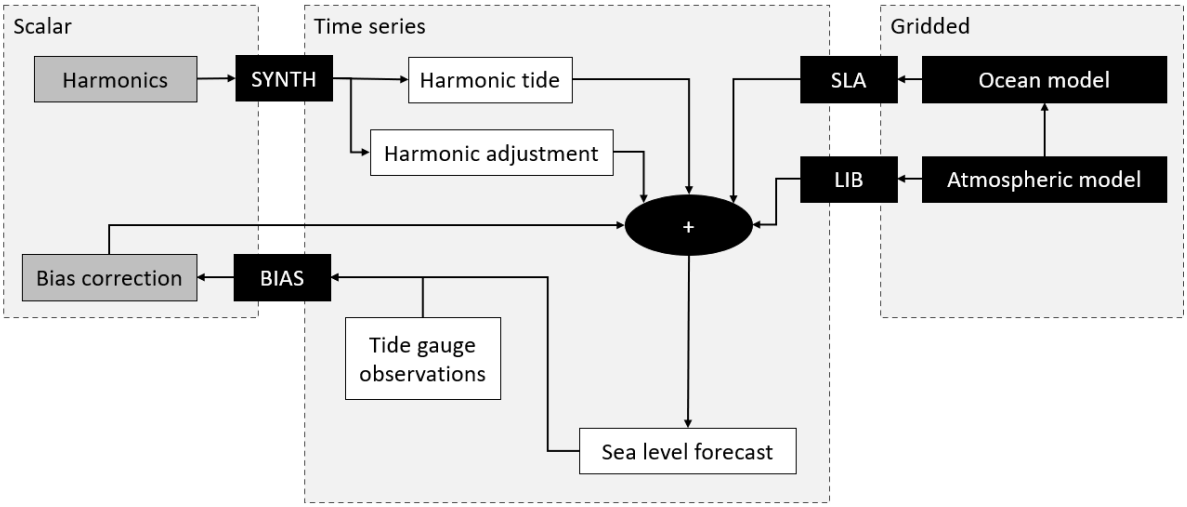

Figure 2: Aggregate Sea Level simplified process flow. White boxes represent data, black represent processes, and grey are ancillary or intermediate datasets. SYNTH indicates tidal synthesis, SLA is sea level anomaly adjustment, LIB is a local inverse barometer approximation, and BIAS is a non-causal filter bias correction scheme.

The suite has approximately 10,000 lines of code, with 132 repository files, around 300 code commits, and with five contributing authors. The codebase is approximately 92% python. Despite its simplicity, the modelling suite ranked a very low SMI of 0.7 in the initial assessment on entering the development environment. Lack of any automated test coverage, outdated scripting languages and unsupported tooling were the main contributors.

Aggregate Sea Level required significant uplift to be able to be accepted for running on the new computing system. In addition to increased documentation, uplift included:

- Increased test coverage, with automated testing in place for many use cases, along with test summary generation;
- A new workflow scheduler, being Cylc 8, with all functionality transferred from the legacy scheduler;
- A major upgrade of the Python version, from Python 2 to Python 3, including code refactoring;
- Uplift to utilising the new package manager;
- A new mechanism for data access, including increased standardisation of data interfaces and de-coupling of some common data handling tasks from the model;
- Adjustment for major differences in the computing environment (such as compilers); and
- A complete tooling overhaul, enabling the model to work within a CI/CD environment.

These changes improved four of the six areas of the SMI: easier to validate, easier to understand, easier to maintain and extend, and improved scientific integrity, and lifted the SMI well above the minimum acceptable value of 1.

The developer migrating the application had no prior knowledge of the current application. In interview they commented on the concerns they felt over where to start with improving test coverage and uplifting code, the difficulties caused by lacks in documentation, the time taken to achieve an acceptable level of testing, and the overall challenges of uplifting code from a low base of test coverage.

4.2 The Dispersion Ensemble Prediction System

The Dispersion Ensemble Prediction System (DEPS) provides 'on-demand' probabilistic model guidance for dispersion of volcanic ash, used for warnings and other products delivered by the Australia-based Volcanic Ash Advisory Centre (Lucas et al., 2018).

The DEPS enables running of the HYSPLIT dispersion model (Stein et al., 2015) coupled to the Bureau's global ensemble numerical weather prediction (NWP) model (named ACCESS-GE) and other internationally-sourced deterministic NWP models. The initial development of the software system was done between mid-2015 and mid-2017.

Since 2019 DEPS has undergone a range of scientific and technological uplifts, to align with better software practices and new computing environments. The suite has approximately 9,000 lines of code, with 717 repository files, over 2000 code commits, and with twenty-six contributing authors. The codebase is 78% python.

Improvements that were in place in a 'version 2' (DEPS2) release prior to this migration include:

- Uplift in testing, including automation;
- Refactoring to improve code quality, including some linting and security features;
- Enhanced features to implement best practice volcanic ash forecast methodologies; and
- Improved use of satellite data for detection and tracking of volcanic ash.

The on-demand workflow (Figure 3) is summarised as:

1. A user inputs model run parameters into a web interface, which prepares a control input file. (Step: Ingest input file);
2. The control input file is delivered to the modelling environment to kick-off the model;
3. The DEPS is executed, with the following steps:
    a. Input data is standardised and ingested (Step: Get observations);
    b. A large number of HYSPLIT models are configured and run (Steps from Analysis setup);
    c. Output skill is analysed and assessed using a Brier score (Brier, 1950) (Step: Calculate skill score 1..N);
    d. Model results are converted to probabilities. (Steps from Filter);
    e. Outputs images and data are produced (Step: Plot).
4. Data is delivered to a user interface and downstream systems.

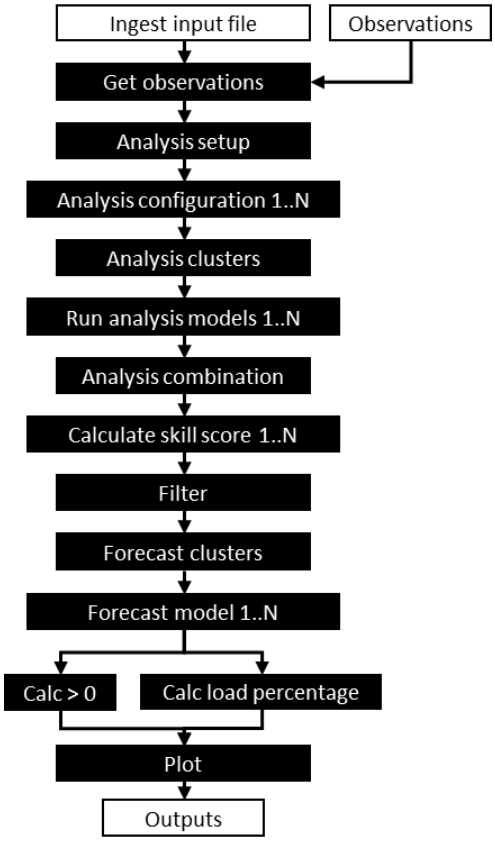

Figure 3: DEPS2 simplified process flow. White boxes represent data and black represent processes

DEPS2 combines external packages, such as HYSPLIT, as well as internally produced code. In addition to automated tests, DEPS2 includes mechanisms for manual inspection of data. The DEPS Output Viewer user interface provides a 'quick look' at the results, to ascertain whether the run has been successful or if further refinement of input parameters is required.

The DEPS2 SMI prior to migration was 2.8, with good testing and good structure, although minor improvements had been flagged for future development. To migrate to the new computing environment the system required a major version update of the Cylc workflow scheduler (to Cylc 8), updates to adopt the new package manager and new data retrieval system, and integration with new CI/CD pipeline. This work was completed in less than four months.

Developers involved with the migration reported that the extensive test coverage on entering the development realm gave them confidence in proper functioning of DEPS2 throughout the migration activity. Errors in configuration, or bugs introduced as part of the rework for the migration, were quickly highlighted by running the unit test suite, and canned data for regression tests gave early confidence of scientific validity of the outputs after upgrading the tooling for the scheduler. Some regression tests failed, due to a slight difference in outputs, and the comparisons had to pass back to scientific experts for evaluation. The outputs were found to be within an acceptable tolerance of the KGOs.

Future work would build this tolerance into the regression tests to improve software quality and remove the requirement for expert re-evaluation of the tests.

In contrast to other software ported by the same developers which had no or limited tests, the porting of this software was deemed straight forward.

On reflection, developers suggested future migration would benefit by adopting the following:

- Uplift to any new CI/CD or other tooling in the current environment and platform;
- Separate science development from technology development;

- Ensure tests implemented are appropriate, and remove any tests not deemed useful for validation purposes;
- Clearly lay out any required remediation work with priorities and deadlines;
- Ensure end-to-end testing of appropriate detail is in place before migration.

4.3 Integrated Model Post-Processing and Verification (IMPROVER)

Statistical post-processing (a form of Machine Learning) increases the accuracy of forecasts of meteorological variables output by multiple models by, for example, accounting for biases, spatial and temporal neighbourhood processing, and automatically adjusting the weighting of inputs from different models. The Integrated Model post-Processing and Verification system (IMPROVER) (Roberts et al., 2023) takes ensemble and deterministic gridded NWP models outputs and produces blended probabilistic fields for use in the forecast delivery pipeline. IMPROVER also provides an extensible framework for future post-processing needs. IMPROVER has over 140,000 lines of code, with just under 1000 repository files, just over 2600 code commits, and with forty-six contributing authors. The codebase is 100% python.

IMPROVER processing steps include:

- Variable adjustments, such as bias correction using analyses and observations
- Conversion of fields to probabilities or percentiles using thresholding
- Neighbourhood processing
- Temporal interpolation
- Reliability calibration
- Combining outputs from individual models to produce blended fields.

A simplified flow-diagram of the IMPROVER internal tasks is shown in Figure 4, including inputs from the Mesoscale Surface Analysis System (MSAS) (Glowacki et al., 2012), which performs the spatial analysis used for calibration of non-rainfall fields.

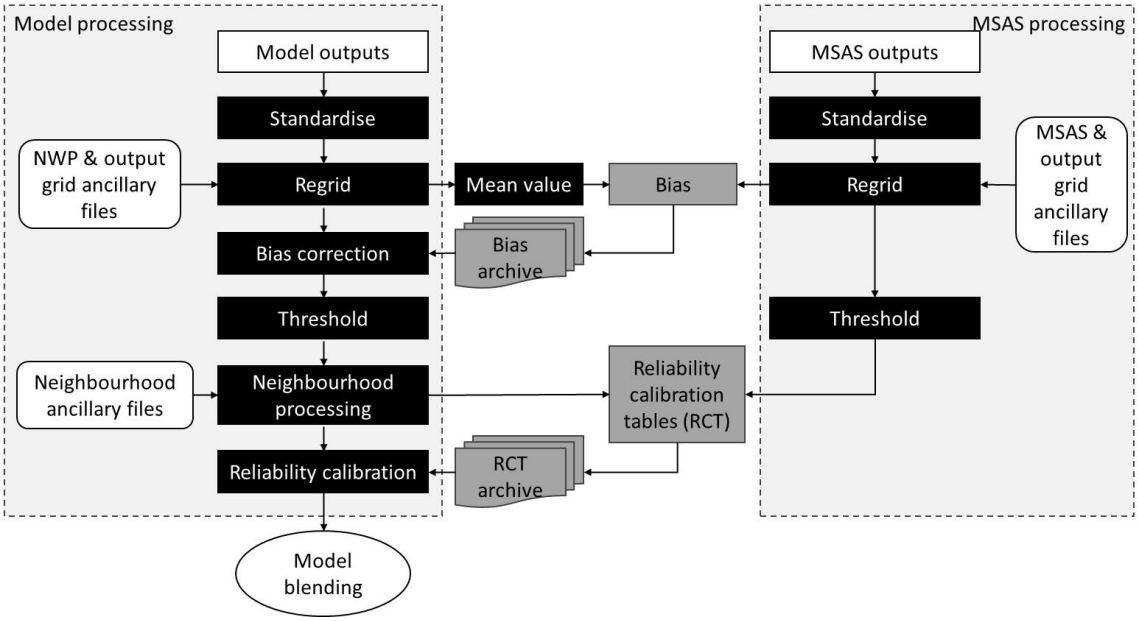

Figure 4: Simplified IMPROVER workflow. White rectangles represent data, black are processes applied to individual models and grey are intermediate datasets. Rounded rectangles are global ancillary data and the white oval represents the final model blending step which combines all individual model outputs

The United Kingdom Meteorological Office (UKMO) started development of IMPROVER in mid-2016, and the Bureau joined the collaboration in late 2019. A strong framework of software quality, rigour and process existed from the inception of the work particularly for code-quality, with Bureau staff contributing notably to an uplift of test quality early on. The system was released internally in the current supercomputing environment in 2022. The initial production release had an SMI of 3.5, meaning that there was nothing needed to improve software

quality, although, as ever, some extra features were identified as 'nice to have'. The most recent release, in late 2023, had an SMI of 3.8, due to an increase in scientific quality.

The high software maturity arose from both technical and process aspects, within an overall test-driven development approach and with a strict peer review format and clear expectations agreed and understood by developers. The software, written in a modern high-level language, has extensive automated tests, with 100 % regression test coverage and 98% unit test coverage, as well as security scanning and linting. A mature Software Quality Assurance Plan outlines manual tests, and system failures are used to inform updates or additions of automated tests. There is a clear separation of concerns within the software, where a decoupler standardises inputs and process-specific code is separate from the main python library. The software maintains compliance with the current target technology stack, and workflow scheduler configurations are built with possible future state productions approaches in mind.

The software was designed to be highly portable and has been successfully deployed to vastly different environments, including HPC mid-range systems and supercomputers, Linux desktops and MacBook. Changes for deployment to our new supercomputing environment are confined to configuration changes to reflect the structure of storage and scaling of data to the platform's processing capability, along with minor scheduler task updates, alignment to a new data retrieval mechanism, and integration with the new CI/CD pipeline.

Developers report that on deployment the install could be readily tested via the various levels of tests. Regression tests use a scientifically verified tolerance on outputs, so do not need modification for new platforms.

Despite the high maturity of the IMPROVER software, the testing of notable science changes cannot be automated. The nature of the product demands months of hindcasts, extensive verification of mass statistics (over spatial and temporal scales) and specific weather event case studies, and manual inspection by forecasters as part of their forecasting process.

Migration of this application is underway and, despite its complexity, the total migration time is estimated to be on the order of one to two months, with effort mostly focussed on optimising performance on the new hardware.

5. A minimum viable good practice approach

The case studies discussed above give an indication of the range of software maturity of applications migrated. A significant proportion of these, particularly older models, had little or no automated testing pre-migration and were running on an old technology stack that was significantly different from the ideal or preferred state. Feedback from developers contained common themes:

1. Software with extensive automated tests, in particular unit and regression tests, reduces unknowns during migration.
2. Mature software with good test coverage and/or aligned with target state tooling was easier and faster to migrate.
3. Existing test quality was highly variable, including tests for specific environments or that did not cover core functions, thus not providing useful tests of the software. Effort required to maintain or transfer these low-quality tests to a new environment likely outweigh any benefit from the testing.
4. Where testing was lacking, regression tests developed in the current operating environment were the most time effective to develop and provided sufficient assurance that scientific validity was maintained.
5. Very good value was gained by developers placing unit tests around complex algorithms or modules known to fail before migration work began.
6. In the absence of adequate existing test coverage developers felt concerned about introducing code changes and were cautious or hesitant in refactoring code due to risks of creating bugs. When a necessary change caused a regression test failure, considerable manual testing and debugging was required, and the complete set of migration tasks took longer than expected.

Generally, the following were identified as highly desirable for future code development:

1. Where possible (i.e. for code within developer control), add unit tests early in development and focus on tests that validate expected scientific results.
2. Where these don't already exist, regression or module level tests for external code (i.e. outside of local control) should be implemented prior to migration.
3. When implementing a significant change to a function that does not have unit tests, write the tests first.
4. No code-merge is allowed to reduce test coverage.
5. Automate tests early in the CI/CD pipeline, if possible, to both reduce manual effort and to prevent poor or troublesome code from being merged.
6. Update software dependencies regularly to the latest versions available.

Our migration efforts also highlighted the importance of good testing versus 'any testing'. Test coverage is often used as a tool for determining code-test quality. However, poor quality tests (e.g. not isolated, environment dependent, or not actually testing code functionality) become a maintenance burden instead of providing assurance of proper software performance. It is critical that as part of any CI/CD implementation a mechanism for evaluating the 'goodness' of test coverage is adopted and applied. Good testing comes from experience. Experience of formal testing across our code migration team ranged from none or little, to decades. Prior to our CI/CD implementation, test coverage for systems without automated unit or regression tests was achieved by manual testing. It has been a steep learning curve for many of our scientists working on code-development to move into the world of unit testing. However, as individuals have experienced the benefits of good test practice, the enthusiasm for it has grown.

Summarising the lessons learned from the migration, and reflecting on the guidance from Staegemann et al. (2023), we identify a minimum-viable approach to CI/CD:

1. For all new internal code development, adopt guidelines on code structure and style and implement these. We recommend the guides at google.github.io > styleguide.
2. Set testing requirements and expectations early, create a Software Quality Assurance Plan before development or migration, and adhere to this plan.
3. Be clear on what constitutes a good test (e.g. code functionality is tested; test operation is independent of the environment). Provide a library of examples and ensure senior software developers with extensive test-development experience are part of the code-change peer-review process.
4. If absent, implement regression tests covering 100% of outputs using KGIs and KGOs before migration or large changes.
5. For software without existing test coverage or for external packages, add unit tests or module level tests to key areas such as critical algorithms and known weak-points. Test coverage should be a minimum of 30% for an SMI score of 1 in that aspect and target a unit test coverage of greater than 80%. Organisational practice varies on acceptable levels of test coverage. These limits (30% and 80%) arose from consideration of experience and risk within our organisation.
6. Automate testing (regression, linting, static security scanning at a minimum; unit testing if available) as part of the CI/CD pipeline as early as feasible.
7. Where automated testing is not deemed sufficient to give confidence in proper function, supplement with manual tests and required test reports until software can be assured as part of integration into the CI/CD pipeline.
8. For large scale changes such as scientific algorithm modification, undertake retrospective forecasts (hindcasts) or long-term testing for assurance.
9. Ensure consistent and clear code-review processes are in place, including peer-review by sufficiently experienced developers on every code-merge.

Table 2 provides a summary of test type, approach, realm (from Figure 1) and realm progression outcome within a CI/CD environment.

Table 2: CI/CD test configuration for MVP for an individual application.

Type of Test	CI/CD or manual	Realm/s	If not met:
Hindcasts/Skill assessment	Manual Performed at key development points	DEV	Do not enter CI/CD pipeline
Unit tests (> 30% coverage)	CI/CD	All	Additional manual testing + test report before realm progression via CI/CD pipeline
Security, static linting	CI/CD	All	Software not progressed to the next realm
Regression tests (target 100 % coverage)	Manual at first, then CI/CD	All	Software not progressed to the next realm
Workflow setup and installation	CI/CD	All	Software not progressed to the next realm
Dependencies and Libraries	CI/CD	All	Software not progressed to the next realm
Disk usage in regularly cycling workflow	Manual at first, then CI/CD	DEV SERVICE	Software not progressed to the next realm
User Acceptance Test	Data from CI/CD deployed software	SERVICE TEST	Do not progress to PROD

In an ideal situation model 'skill' (or accuracy) evaluation would be automated and part of the CI/CD pipeline. However, many scientific software applications need extended periods of scientific validation, with both automated and manual interrogation. Similarly, despite data being delivered automatically from the TEST realm, testing by users in downstream and external systems is largely manual.

To allow for these manual processes we propose two 'gateways' to software progression through the realms:

1. From DEV to SERVICE: This gateway ensures that requirements for scientific skill have been met where a significant change to the software has been made. Evaluation is made via a standardised test report that summarises the scientific skill, the computing resources used and the timeliness of delivery of outputs. Peer review of this report is done by a panel of knowledgeable staff across scientific subject matter experts, research-to-operations leads and ICT operations and support areas.

   Once Gate1 is first cleared, any small, incremental changes not affecting the scientific aspects of the outputs should be progressed automatically via CI/CD and the internal code-review process.

2. From TEST to PROD: This gateway ensures downstream users have signed off system updates as well as serving as a manual review process for low-maturity software. Evaluation is again made via a standardised report that summarises the changes made, any effects on downstream systems and business user readiness for the change. Peer review of this report is conducted by a panel of knowledgeable staff across research-to-operations, ICT operations and support areas and business owners.

Figure 5 summarises the flow for the proposed minimum viable process, taking account of both high and low maturity applications. The process flow reinforces continued improvement of software maturity with code-reviews and gateways ensuring that added or refactored code is appropriately tested. For applications with low-maturity or high-complexity, additional manual assessments are mandated prior to progression via CI/CD controlled realms. The benefits of this include quality assurance for the model going to production via Gate 2 as well as protection of other models in production.

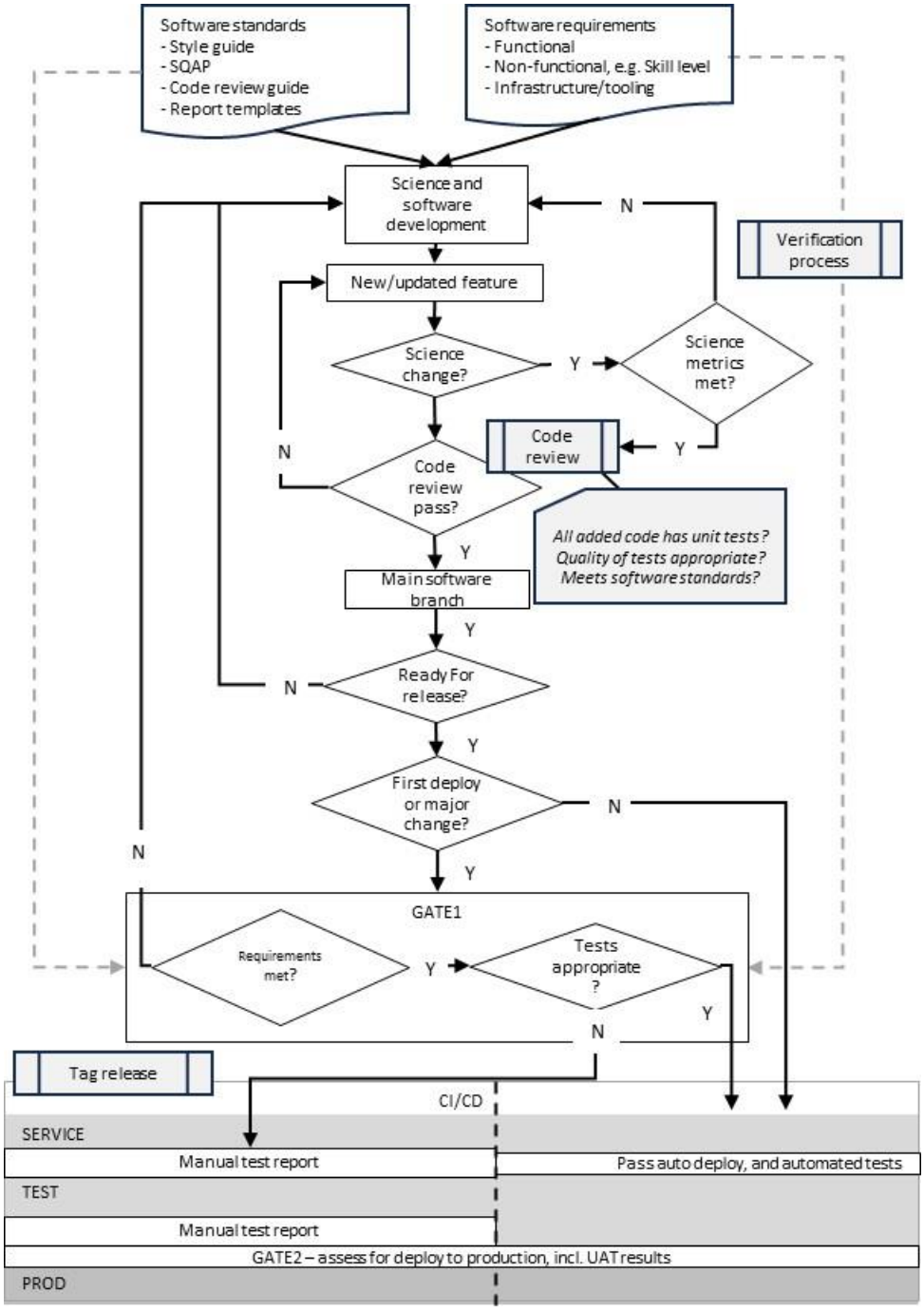

Figure 5: Overall minimum viable process flow. This process encourages continued improvement of software maturity and uses added manual processes to assess significant change or low-maturity applications

6. Conclusion

Planning and implementation of good software testing has been increasingly applied in scientific software development over recent years, with consequential improvements in software quality. This paper explores the nature of scientific software quality in a constrained production environment, where software must be sufficiently mature to cope with everyday operational challenges such as missing data, unintended user inputs, variable data quality, and values outside the bounds of those previously tested. The specific problem situation presented here arose in the migration of software from an existing computing environment into a new one, where adoption of formal CI/CD methodology and tooling required testing sufficient for the software to pass to production through a formal research-to-operations pathway, across multiple computing realms. The problem

situation was further complicated by the community nature of scientific software development, with significant code sharing across international boundaries and many models being formed from a mix of externally and internally developed code, each with differing opportunities and methods for developers to contribute to increased code quality.

Investigation of three case studies of software migration, along with the reflections of developers working on this migration and consideration of the broader applicability of our findings to other situations, delivered a minimum viable good practice approach for application to scientific environmental modelling. By establishing a clear set of expectations and processes that increase software maturity with every code increment and deployment, the overall quality of applications running in production are improved and various matters of developer wellbeing are also addressed.

Acknowledgements

The authors gratefully acknowledge the interview responses and reflections of the developers who contributed to this review, including Dr Gemma Lloyd, Dr Nicholas Leerdam, Sean Loh, Dr Daehyok Shin, Daniel Karney, Dr Christopher Down, Dr Timothy Hume, Dr Andy Thomas and Dr Tom Gale. We also thank internal reviewers for the contributions to the quality of this paper.

References

Arvanitou, E. M., Ampatzoglou, A., Chatzigeorgiou, A., & Carver, J. C. (2021). Software engineering practices for scientific software development: A systematic mapping study. Journal of Systems and Software, 172. https://doi.org/10.1016/j.jss.2020.110848

Beck, K. (2002). Test Driven Development. By Example (Addison-Wesley Signature). Addison-Wesley Longman, Amsterdam.

Brier, G. W. (1950). Verification of forecasts expressed in terms of probability. Monthly Weather Review, Vol. 78, No. 1, pp. 1–3.

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., & Horowitz, B. M. (1999). Model-based testing in practice. Proceedings of the 21st International Conference on Software Engineering, 285–294.

dos Santos, J., Martins, L. E. G., de Santiago Júnior, V. A., Povoa, L. V., & dos Santos, L. B. R. (2020). Software requirements testing approaches: a systematic literature review. Requirements Engineering, 25(3). https://doi.org/10.1007/s00766-019-00325-w

Dubois, P. F. (2012). Testing Scientific Programs. Computing in Science and Engg., 14(4), 69–73. https://doi.org/10.1109/MCSE.2012.84

Glowacki, T. J., Xiao, Y., & Steinle, P. (2012). Mesoscale Surface Analysis System for the Australian Domain: Design Issues, Development Status, and System Validation. Weather and Forecasting, 27(1), 141–157. https://doi.org/https://doi.org/10.1175/WAF-D-10-05063.1

Gupta, A., Poels, G., & Bera, P. (2022). Using Conceptual Models in Agile Software Development: A Possible Solution to Requirements Engineering Challenges in Agile Projects. IEEE Access, 10. https://doi.org/10.1109/ACCESS.2022.3221428

Heaton, D., & Carver, J. C. (2015). Claims about the use of software engineering practices in science: A systematic literature review. Information and Software Technology, 67, 207–219. https://doi.org/https://doi.org/10.1016/j.infsof.2015.07.011

Iwanaga, T., Rahman, J., Partington, D., Croke, B., & Jakeman, A. J. (2018). Software Development Best Practices for Integrated Model Development. Proceedings International Conference on Environmental Modelling and Software, Fort Collins, Colorado https://doi.org/10.13140/RG.2.2.27034.34247

Jakeman, A. J., Elsawah, S., Wang, H.-H., Hamilton, S. H., Melsen, L., & Grimm, V. (2024). Towards normalizing good practice across the whole modeling cycle: its instrumentation and future research topics. Socio-Environmental Systems Modelling, 6, 18755.

Kanewala, U., & Bieman, J. M. (2014). Testing scientific software: A systematic literature review. In Information and Software Technology (Vol. 56, Issue 10). https://doi.org/10.1016/j.infsof.2014.05.006

Kanewala, U., & Chen, T. Y. (2019). Metamorphic Testing: A Simple Yet Effective Approach for Testing Scientific Software. Computing in Science & Engineering, 21(1), 66–72. https://doi.org/10.1109/MCSE.2018.2875368

Lucas, C., Potts, R. J., Zidikheri, M. J., Dare, R. A., Manickam, M., Wain, A., & Bear-Crozier, A. (2018). Improvements in the Detection and Prediction of Volcanic Ash for Aviation at the Australian Bureau of Meteorology. 98th American Meteorological Society Annual Meeting, 6pp.

Nanthaamornphong, A., & Carver, J. C. (2017). Test-Driven Development in scientific software: a survey. Software Quality Journal, 25(2). https://doi.org/10.1007/s11219-015-9292-4

Peng, Z., Lin, X., Simon, M., & Niu, N. (2021). Unit and regression tests of scientific software: A study on SWMM. Journal of Computational Science, 53. https://doi.org/10.1016/j.jocs.2021.101347

Pereira, N. S., Lima, P., Guerra, E., & Meirelles, P. (2022). Towards Automated Playtesting in Game Development. Proceedings of SBGames 2021. https://doi.org/10.5753/sbgames_estendido.2021.19666

Roberts, N., Ayliffe, B., Evans, G., Moseley, S., Rust, F., Sandford, C., Trzeciak, T., Abernethy, P., Beard, L., Crosswaite, N., Fitzpatrick, B., Flowerdew, J., Gale, T., Holly, L., Hopkinson, A., Hurst, K., Jackson, S., Jones, C., Mylne, K., … Worsfold, M. (2023). IMPROVER: The New Probabilistic Postprocessing System at the Met Office. Bulletin of the American Meteorological Society, 104, E680–E697. https://doi.org/10.1175/BAMS-D-21-0273.1

Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. In IEEE Access (Vol. 5). https://doi.org/10.1109/ACCESS.2017.2685629

Staegemann, D., Volk, M., Pohl, M., Haertel, C., Hintsch, J., & Turowski, K. (2023). Identifying Guidelines for Test-Driven Development in Software Engineering—A Literature Review. Lecture Notes in Networks and Systems, 465. https://doi.org/10.1007/978-981-19-2397-5_30

Stein, A. F., Draxler, R. R., Rolph, G. D., Stunder, B. J. B., Cohen, M. D., & Ngan, F. (2015). NOAA's HYSPLIT Atmospheric Transport and Dispersion Modeling System. Bulletin of the American Meteorological Society, 96(12), 2059–2077. https://doi.org/https://doi.org/10.1175/BAMS-D-14-00110.1

Taylor, A., & Brassington, G. B. (2017). Sea Level Forecasts Aggregated from Established Operational Systems. In Journal of Marine Science and Engineering (Vol. 5, Issue 3). https://doi.org/10.3390/jmse5030033

Uzun, B., & Tekinerdogan, B. (2018). Model-driven architecture based testing: A systematic literature review. Information and Software Technology, 102. https://doi.org/10.1016/j.infsof.2018.05.004