We would like to thank the reviewers for careful consideration of our work and for providing thoughtful feedback that has improved the quality of the manuscript in many ways, including the documentation, the detail and value of the examples, and the clarity of the discussion about automatic differentiation. We have numbered the comments from each reviewer below. Responses are in blue with quoted text from the new tracked-changes manuscript in *italics* with associated line numbers.

Reviewer 1

1. General: given the similarities and the shared foundation of SPEEDY, I think it would be good to cite the SpeedyWeather.jl project somewhere (I am not an author of this model incidentally).

   Thank you for pointing out this oversight. We do see SpeedyWeather as an inspiration for this work and should have included a formal citation.

   Updated text (lines 67–69): *"This work builds upon a growing movement to generate modular, differentiable Earth system components in modern programming languages with GPU acceleration (Klöwer et al., 2024; Häfner et al., 2018; Fang et al., 2024; Wagner et al., 2025)."*

2. Section 2.1, Dynamical Core: could you say a word about the limits of scalability of the JCM dycore? What determines the limit of T425? Is it the memory of the current generation of accelerators? Would it be feasible to do domain decomposition like traditional models, in order to run on distributed memory clusters, and therefore access higher resolutions? Or was it decided a priori that JCM will not target those resolutions for simplicity?

   Thanks for the question, we have clarified the existing technical obstacles and the fact that the current configuration is intended for global studies.

   Added text (lines 89–93): *"There are no technical obstacles to adding support for resolutions beyond T425, though other features may be of higher priority for high-resolution studies, such as support for more vertical levels or for a non-spectral dynamical core. (Spectral integration schemes must simulate the entire globe, making them inefficient for region-focused experiments, and the spectral transforms required at each timestep eventually bottleneck performance as they scale worse than linearly with the number of grid points.)"*

3. Line 112: typo - "parametrization" or "parameterization".

   Updated text (lines 118–119): *parameterization*

4. Section 3, Model Interface (API): nothing wrong with code listings, but it's neater to treat them like any other figure and make an explicit reference to them in the text, e.g. "Figure X gives an example usage of JCM".

Code listings have been converted to figures, and references to them have been added. In Section 3:

Added text (lines 171–172): *Example code for running the model is shown in Figure 1.*

In Section 6.3:

Added text (lines 337–339): *The code block in Figure 9 implements the described experiment, calculating the gradient of f(*longwave_rad.ftop*) with respect to the open-ocean surface albedo* Parameters.albsea *using JVP (*jax.jvp*).*

5. Section 5, Performance: I would recommend presenting all computational speed results in common units of e.g. "simulated years (or days) per day" (SYPD/SDPD). This gives a nice intuitive number which summarises what the user cares about.

Original Text: "Figure 5 shows the total time to run a default configuration of JCM at T85 (approx. 150km resolution at the Equator) using either a laptop CPU, laptop GPU (NVIDIA RTX 4050), or Google Cloud TPU. The behavior in the figure has been our typical experience: runtime on GPUs and TPUs is at least an order of magnitude faster than on CPUs (except for Apple Silicon chips), with parallelization (sharding the model state arrays so they can be distributed among multiple devices) giving a significant speedup to the TPU runs on Google Cloud. For this set of parameters, the laptop GPU was faster than the cloud TPUs, but initial testing indicates that the TPU (like any high-performance cloud platform) will scale better for higher resolutions than the laptop GPU with its limited VRAM. The platforms used here do not all have the same number of cores or clock speeds; the goal is to illustrate typical runtime on common devices.

As another point of comparison, JCM on the RTX 4050 laptop GPU was about 40% faster per year of model time than speedy.f90 on the same laptop's Intel i9 13th Gen CPU.

[...]

With regards to the model run configuration itself, the number of operations for a model run typically increases with the square of the spectral truncation, with an additional factor for reductions to the model timestep (sometimes needed for stability at a higher resolution). However, runtime is observed to scale better than the number of operations, presumably due to parallelization. Figure 6 shows how model runtime scales with the spatial resolution of the model grid, using as a benchmark a one year simulation with default 30 minute time step, run on a NVIDIA RTX 4050 laptop GPU. At T31 resolution, one year of model time takes only a minute and a half, and even T119 can be run for a year in less than 10 minutes."

Updated Text (lines 223–240): *"Figure 6 shows the simulation speed, in units of simulated years per day (SYPD), of running JCM at various resolutions and on various device types. The behavior in the figure has been our typical experience: runtime on GPUs and TPUs is significantly faster than on CPUs, with parallelization (sharding the model state arrays so they can be distributed among multiple devices) giving an additional significant speedup to TPU runs on Google Cloud (parallelization was not used for the results in Figure 6). The dashed line in the figure represents the theoretical quadratic scaling of number of operations (and memory requirement) with spectral truncation.*

[...]
It should be noted that in practice, the runtime for the higher resolutions may pick up another factor of 2 or 3 relative to lower resolutions due to reductions to the model timestep that are eventually needed for stability. For the results shown in Figure 6, the timestep was left as the default 30 minutes for all resolutions."

6. Figure 5: the presentation of the benchmark here is a little unusual. The graph shows elapsed time vs. simulation time. I would imagine that all of these lines are in fact straight when plotted on a linear scale, indicating that the wall time per step is constant, assuming each step is identical (e.g. no I/O on every nth step). Then, why bother with a graph at all? All that matters is the gradient - the speed of the simulation in simulated-things-per-actualthing, which can just be presented in a table. I would recommend that approach.

   By putting performance results in terms of simulated years per day, the new Figure 6 covers this information, so Figure 5 has been removed entirely.

7. Figure 6: could you mention in the caption what hardware you used?

   Updated Figure 6 caption: *"Model performance on different hardware for a default JCM run, 30 minute timestep. To generate this plot, runs of 1 simulated year were conducted on 3 types of Google Cloud virtual machines: a CPU VM (with 32x Intel Xeon Platinum 8581C), a GPU VM (with a 16GB Tesla P100 PCIe), and a TPU VM (with a Google Cloud TPU v5 lite). These devices were chosen as comparable based on having similar pricing (approx. $1.50 / hour) at time of writing."*

8. Line 241: I would be a bit more careful in wording here. ECMWF's IFS has hand-coded tangent-linear and adjoint for its entire atmospheric model, and has maintained them for 25 years now. Maintaining these parallel codes is of course a substantial burden, but far from "intractable".

   Thank you for the advice, we have updated the text accordingly.

   Updated Text (lines 250–252): *"Given the complexity of state-of-the-art Fortran models, this manual generation of gradients is expensive to implement and maintain. Thus, in most cases, gradient information is inaccessible or presents a substantial burden to model developers."*

9. Section 6.1, Automatic-differentiation (VJP and JVP): personally I would find this explanation much easier with a few equations. The verbal description left a few unanswered questions for me. For example, what is the mathematical difference between VJP and JVP? If I'm understanding the text correctly, one is simply the transpose of the other.

   This is a good question and we have updated the text to be more clear about the mathematical equivalence. While the TLM and adjoint operators are transposes of each other, the operations JVP and VJP are not. Importantly, VJP and JVP do not evaluate the whole Jacobian (it would be very expensive), they evaluate the relevant row or column.

Updated Text (lines 253–264): *"JAX provides JCM with its AD capabilities (Bradbury et al., 2018). In JAX, gradients can be computed in either forward (tangent linear) or reverse (adjoint) mode. Each mode has multiple use cases and can be applied to a wide variety of science problems. We use F(x) to denote the tangent linear model of a function F(x). The relationship of F(x) to the adjoint $F^T(x)$, where both are linearized about x, is*

$$\langle F(x)v, w \rangle = \langle v, F^T(x)w \rangle , \quad (1)$$

where v is the tangent vector, w is the co-tangent vector, and $\langle \cdot \rangle$ denotes the inner product. Forward-mode differentiation is applied through the Jacobian vector product (JVP) and is equivalent to F(x)v. This evaluates the multiplication of a user-specified tangent vector, v, with the tangent linear model, F(x). Reverse-mode differentiation evaluates a vector-Jacobian product (VJP), equivalent to $F^T(x)w$. This multiplies a user-specified co-tangent vector, w, with the AD-derived adjoint, $F^T(x)$, of the function F(x). Both VJP and JVP avoid computing the entire Jacobian of the function to minimize computational cost, but they can be used to reconstruct it if necessary. Example applications are discussed in more detail in the following sections."

Also, when you say such and such is "computationally efficient" under certain conditions, what is meant exactly?

Original Text: "VJP is computationally efficient when the output space of the function is much smaller than the input space… In contrast to VJP, JVP is computationally efficient when the output dimension is larger than or equal to the input dimension of a function."

Updated Text (lines 265–272): *"In large-scale geophysical models integrated over long time windows, JVP typically costs approximately 1–2 times the cost of a nonlinear forward model integration. In contrast, VJP often costs approximately 3–6 times the cost of the forward integration, depending on memory constraints, checkpointing strategy, and solver structure (Heimbach et al., 2005; Evensen et al., 2020). Reverse-mode differentiation is more computationally efficient than forward mode for gradient evaluation when the output dimension of the function of interest is much smaller than the input dimension (e.g., a scalar loss function). In this case, a single VJP evaluation produces the full gradient, whereas forward mode would require n separate JVP evaluations, where n is the input dimension. Conversely, forward-mode differentiation is more efficient when the number of inputs is much smaller than the number of outputs."*

10. Section 6.2, Calibration: could you be specific about what "statistics" you compute in this example?

    We updated the calibration example to be over two parameters (see Reviewer 2, question 6). We have also added the following sentence to clarify the statistics used in this problem

Updated Text (lines 285–288): "*We generate synthetic observed statistics by running the JCM in the default aquaplanet configuration, which has values of stratiform cloud albedo = 0.5 and relative humidity threshold = 0.3. The calculated statistics are the spatial and temporal average of each model output variable over a 5-day model run."*

11. Line 311-: again, this code listing would be better as a figure, though this one at least has a reference in the text. Why does the line number start at 13 by the way?

The code listing has been converted to a figure, and code has been added to make it self-contained (the line number previously began at 13 because the code used variables from the prior code listing).