Diagnostic Assessment and Theoretical Reconstruction of Tropical Cyclone Genesis in Simplified Non-Hydrostatic Models

1 Introduction: The Challenge of Minimalist Atmospheric Modeling

The computational simulation of tropical cyclones (TCs) represents one of the most complex challenges in geophysical fluid dynamics, requiring the harmonious integration of non-linear advection, thermodynamic phase changes, and boundary layer turbulence. While operational models like the Weather Research and Forecasting (WRF) model or the Model for Prediction Across Scales (MPAS) employ millions of lines of code to handle these interactions, there is significant scientific value in "toy models" or idealized simplified models. These minimalist frameworks allow researchers to isolate specific mechanisms, such as the Wind-Induced Surface Heat Exchange (WISHE) instability or the dynamics of the eyewall mesovortices, without the confounding noise of complex parameterizations.

However, the reduction of the Navier-Stokes equations to a simplified form—specifically the incompressible Euler or Navier-Stokes equations often used in basic Computational Fluid Dynamics (CFD) solvers—introduces a critical disconnect when applied to atmospheric phenomena. The user's current project, a Python-based hurricane simulation, has encountered a fundamental failure mode: the simulated storm initializes with a defined vortex structure but fails to intensify, lacks vertical motion despite thermal anomalies, and decays monotonically due to friction. The specific observation that "warm, moist surface air just sits there" is a pathognomonic sign of a decoupled thermodynamic-dynamic system. In the current architecture, where vertical velocity (w) is derived solely from the pressure Poisson solver to enforce incompressibility ($\nabla \cdot \mathbf{u} = 0$), the fluid acts as a neutral-density medium. The pressure solver in an incompressible framework acts purely to redistribute pressure instantaneously to maintain mass continuity; it does not inherently account for density variations unless explicitly forced by a buoyancy term.

This report provides an exhaustive theoretical and practical analysis of this failure. It dissects the limitations of the "Warm-Core Initialization" (Option D) proposed by the user, demonstrating why thermodynamic potential energy cannot be converted to kinetic energy in the absence of the Boussinesq approximation (Option C). Furthermore, it outlines the implementation of a complete physics package—including active buoyancy, saturation adjustment (simple microphysics), and surface flux parameterizations—required to transform the code from a kinematic fluid exercise into a physically robust atmospheric model capable of sustaining a tropical cyclone.

2 Theoretical Framework of Tropical Cyclone Dynamics

To understand why the current model fails, one must first revisit the fundamental governing equations of atmospheric motion and identifying where the simplified CFD approach diverges from meteorological reality.

2.1 The Limitations of Incompressible Flow in Hurricane Modeling

The incompressible Navier-Stokes equations, which form the backbone of many standard CFD solvers (including the one likely used by the user), describe a fluid of constant reference density ρ_0. The momentum and continuity equations are typically written as:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho_0}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{F} \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

In this system, the pressure p is not a thermodynamic variable defined by an equation of state (such as the ideal gas law $p = \rho RT$). Instead, pressure acts as a Lagrange multiplier—a mathematical artifact whose sole purpose is to enforce the divergence-free constraint on the velocity field. The "Projection Method" or "Chorin's Splitting," widely used in such solvers, reinforces this strictly kinematic role. The solver computes a tentative velocity field $\mathbf{u}^*$ based on advection and diffusion, and then solves a Poisson equation for pressure:

$$\nabla^2 p = \frac{\rho_0}{\Delta t}\nabla \cdot \mathbf{u}^* \tag{3}$$

The gradient of this pressure is then used to correct the velocity, ensuring $\nabla \cdot \mathbf{u}^{n+1} = 0$.

The critical flaw in applying this unmodified framework to hurricane genesis is the interpretation of the temperature field. In the user's current model, temperature T is likely treated as a passive scalar, governed by an advection-diffusion equation:

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \kappa \nabla^2 T \tag{4}$$

In this configuration, the temperature field evolves based on the wind, but the wind does not respond to the temperature. A warm bubble (like the +5°C anomaly in Option D) creates no buoyancy force because the momentum equation contains no term linking ρ or T to gravitational acceleration. The pressure solver sees a region of fluid that is kinematically identical to its surroundings. If the initial velocity is zero, the advection term is zero, and the pressure gradient adjusts simply to balance whatever external forces exist. If gravity is not coupled to density anomalies, the net force is zero. Thus, the warm air "sits there" because the mathematics contain no instruction for it to rise.

2.2 The Boussinesq Approximation: Restoring Vertical Connectivity

Real atmospheric flow is compressible; density ρ varies with pressure and temperature. However, solving the fully compressible Euler equations requires resolving acoustic waves, which travel at the speed of sound ($c_s \approx 340$ m/s). This necessitates extremely small time steps to satisfy the Courant-Friedrichs-Lewy (CFL) condition ($C = u\Delta t/\Delta x < 1$). For a convective model where wind speeds are ~ 50 m/s, the acoustic constraint makes the simulation prohibitively expensive for long integrations (like the 300k frames mentioned).

To bridge the gap between incompressible efficiency and compressible realism, atmospheric models operating at the mesoscale (such as cloud-resolving models or idealized hurricane simulations) employ the Boussinesq Approximation. This approximation fundamentally alters the momentum equation by assuming that density variations are negligible in the inertial terms (mass $\times$ acceleration) but crucial in the buoyancy term (density $\times$ gravity).

The vertical momentum equation transforms from the generic Navier-Stokes form to:

$$\frac{Dw}{Dt} = -\frac{1}{\rho_0}\frac{\partial p'}{\partial z} + b + F_z \tag{5}$$

Here, p' represents the perturbation pressure from the hydrostatic mean, and b is the buoyancy acceleration. Using the linearized equation of state for an ideal gas, buoyancy is defined as:

$$b = -g\frac{\rho - \rho_0}{\rho_0} \approx g\frac{T - \bar{T}(z)}{\bar{T}(z)} \tag{6}$$

where $\bar{T}(z)$ is the background vertical temperature profile. When moisture is included, the virtual temperature T_v (which accounts for the lower density of water vapor compared to dry air) and liquid water loading (the drag exerted by raindrops) are incorporated:

$$b = g\left(\frac{T'}{\bar{T}} + 0.61q_v' - q_l\right) \tag{7}$$

By implementing this term, the "passive" temperature scalar becomes an "active" dynamic driver. A warm anomaly ($T' > 0$) generates a positive vertical acceleration ($b > 0$). This increases the vertical velocity w. The divergence of the provisional velocity field becomes positive ($\nabla \cdot \mathbf{u}^* > 0$) in the column. The pressure solver then reacts to this divergence, establishing a non-hydrostatic pressure gradient that enforces continuity by drawing air in at the bottom (convergence) and expelling it at the top (divergence). This establishes the secondary circulation—the "in-up-out" flow—that is the engine of the tropical cyclone.

2.3 The WISHE Mechanism: Sustaining the Vortex

The user reports that the storm "steadily decays into lower than 10 knots." This suggests that the model is suffering from energy starvation. A tropical cyclone is a dissipative structure; it constantly loses kinetic energy to surface friction and turbulent dissipation. In an "initial value problem" where a vortex is initialized without a continuous energy source, the system will inevitably spin down as friction drains its initial angular momentum.

The maintenance and intensification of real tropical cyclones rely on the Wind-Induced Surface Heat Exchange (WISHE) mechanism. The theory posits that the energy source for the storm is the thermodynamic disequilibrium between the ocean surface (warm and saturated) and the boundary layer air (cooler and subsaturated). Crucially, the rate of enthalpy transfer is a function of wind speed:

$$F_k = C_k \rho |\mathbf{u}_{\text{sfc}}|(k_s^* - k_{\text{air}}) \tag{8}$$

where k is the specific enthalpy of air and k_s^* is the saturation enthalpy at the sea surface temperature (SST).

This creates a positive feedback loop:

1. The vortex initializes, generating surface winds.
2. Stronger winds increase the evaporation rate (latent heat flux) from the ocean.
3. This moisture is transported inward and upward into the eyewall.
4. Latent heat release in the eyewall warms the core, increasing buoyancy.
5. Increased buoyancy lowers the central surface pressure.
6. The pressure gradient tightens, further increasing surface winds.

Without the implementation of surface flux parameterizations that explicitly link evaporation to $|\mathbf{u}|$, the simulated hurricane is effectively cut off from its fuel source. It consumes the moisture provided in the initial condition (the warm core) and then dies. The decay observed by the user is physically correct for a closed system with friction; the error lies in treating the hurricane as a closed system rather than an open one coupled to the ocean.

3 Diagnostic Analysis of the "Warm-Core Initialization" (Option D)

The user asks to evaluate "Option D" before implementing "Option C" (Boussinesq). Option D involves initializing the atmosphere at a warmer base state (25°C vs 15°C) and adding a 5°C warm anomaly.

3.1 Thermodynamic Necessity vs. Dynamic Insufficiency

The Thermodynamic Argument (Pro-Option D): The proposal to increase the base temperature from 15°C to 25°C is thermodynamically sound and necessary. The capacity of the atmosphere to hold water vapor is governed by the Clausius-Clapeyron relation, which is non-linear (exponential).

- At 15°C, the saturation vapor pressure $e_s \approx 17.0$ hPa.
- At 25°C, $e_s \approx 31.7$ hPa.

By shifting the base state to 25°C, the model nearly doubles the potential latent heat energy available ($L_v q_{\text{sat}}$) for the storm. A hurricane simulated in a 15°C environment would be incredibly weak, akin to a polar low, because the "fuel density" of the air is too low to support a deep warm core against adiabatic cooling. Therefore, Option D is a pre-requisite for a realistic simulation.

The Dynamic Argument (Anti-Option D as Standalone): However, the user's hope that this change will "fix" the lack of rising motion is misplaced. In the current incompressible architecture (without Boussinesq terms), the temperature field is dynamically decoupled.

Pressure Solver Invariance: The pressure Poisson equation $\nabla^2 p = S$ depends on the source term S, which is a function of velocity divergence. If the velocity field is initially zero (or purely rotational), and there is no buoyancy term in the momentum equation, the pressure field essentially adjusts to maintain that state. The absolute value of the temperature (15 vs 25) or the gradient of temperature (the warm anomaly) does not appear in the standard incompressible momentum equation.

Static Result: Consequently, the code will initialize a warmer, moister atmosphere that is statically unstable in reality but neutrally stable in the model's "eyes." The warm moist air will indeed "just sit there" because the instruction set governing its motion (the Navier-Stokes solver) lacks the line of code that says "if hot, then rise."

3.2 The Hydrostatic Paradox in Incompressible Solvers

The user mentions an "immediate low pressure anomaly" expected from the warm core. In a hydrostatic atmosphere, pressure is the weight of the column above:

$$p(z) = \int_z^\infty \rho g \, dz \tag{9}$$

Since $\rho \propto 1/T$, a warm column weighs less, creating a surface low.

However, in a non-hydrostatic projection solver, pressure is a dynamic variable derived from the continuity constraint. While one can initialize the pressure field hydrostatically, the very first time step of the solver will "re-project" the pressure. If the divergence is zero (because no air is moving yet), the solver might effectively wipe out the hydrostatic low if it conflicts with the boundary conditions or the divergence-free constraint, unless the buoyancy force is active to sustain that vertical pressure gradient dynamic balance.

Conclusion: Option D provides the necessary potential energy (fuel), but Option C (Boussinesq) provides the engine to burn it. Implementing D without C is futile.

4 Comprehensive Implementation Strategy

To reconstruct the model into a functional hurricane simulation, the codebase must be augmented with three active physics modules: The Boussinesq Corrector, The Microphysics Scheme, and The Surface Flux Scheme. The following sections detail the mathematical formulation and Python implementation for each.

4.1 Phase 1: The Boussinesq Buoyancy Term (The Engine)

This is the non-negotiable fix for the "stationary air" problem. The vertical velocity w must be updated explicitly with a buoyancy acceleration before the pressure projection step.

Mathematical Formulation: The buoyancy acceleration a_b is applied to the vertical component of the velocity field.

$$a_b(x, y, z, t) = g \left(\frac{T(x, y, z, t) - \bar{T}(z)}{\bar{T}(z)} \right) \tag{10}$$

where $\bar{T}(z)$ is the horizontal mean temperature at height z. Subtracting the mean is crucial to remove the hydrostatic balance component, allowing the solver to handle only the deviations (perturbations) that drive flow.

Python Implementation Details: Assuming the model uses numpy (or cupy for GPU, denoted as xp), and the velocity array u has shape (3, nz, ny, nx) (z, y, x ordering is common in meteorology, though x, y, z is common in CFD; the code below assumes (nz, ny, nx) spatial dimensions).

```python
def apply_buoyancy(self, dt):
    """
    Applies Boussinesq buoyancy to the vertical velocity
       component.
    Should be called after advection and before pressure
       projection.
    """
    g = 9.81  # Gravity m/s^2

    # 1. Calculate the Horizontal Mean Temperature Profile (
       Reference State)
    # Assuming self.T is shape (nz, ny, nx). Axis (1,2) are
       horizontal.
    T_ref = xp.mean(self.T, axis=(1, 2))

    # 2. Reshape T_ref for broadcasting against 3D array
    # T_ref_3d will have shape (nz, 1, 1)
    T_ref_3d = T_ref[:, None, None]

    # 3. Calculate Temperature Perturbation
    theta_prime = self.T - T_ref_3d

    # 4. Calculate Buoyancy Acceleration
    # B = g * (T' / T_ref)
    buoyancy = g * (theta_prime / T_ref_3d)

    # 5. Apply to Vertical Velocity
    # Assuming self.u has shape (3, nz, ny, nx) and index 0 is z-
       velocity (w)
    # Or if separate arrays u, v, w:
    self.w += buoyancy * dt

    # NOTE: If using a staggered grid (C-grid), w is defined at
       cell faces (k+1/2).
    # T is at cell centers (k). You must interpolate buoyancy to
       w-faces.
    # buoyancy_face = 0.5 * (buoyancy[:-1] + buoyancy[1:])
    # self.w[1:-1] += buoyancy_face * dt
```

Grid Considerations: If the user is using a Collocated Grid (all variables at cell centers), applying buoyancy is straightforward but can lead to "checkerboard" pressure oscillations. If using a Staggered Grid (Arakawa C), which is standard for atmospheric models, w lives on the vertical faces of the cells, while T lives at the center. The buoyancy term must be averaged from the centers to the faces: $b_{k+1/2} = 0.5(b_k + b_{k+1})$. This slight averaging acts as a filter and stabilizes the vertical motion.

4.2 Phase 2: Thermodynamics and Microphysics (The Fuel Injector)

Once the air rises due to buoyancy, it cools adiabatically.

$$\frac{DT}{Dt} \approx -w\frac{g}{c_p} \tag{11}$$

In a standard atmosphere, this cooling rate ($\sim$ 9.8 K/km) quickly makes the rising parcel colder than the environment, creating negative buoyancy that stops the updraft. The storm dies. To sustain the updraft, Latent Heat Release is required. As the parcel cools to its dew point, water vapor condenses. The release of latent heat ($L_v \approx 2.5 \times 10^6$ J/kg) reduces the cooling rate to the moist adiabatic lapse rate ($\sim$ 6.5 K/km). This keeps the parcel warmer than the environment for a much greater depth, powering the deep convection of the eyewall.

The "Simple Kessler" Scheme: The user requires a minimal microphysics implementation. The Kessler (1969) scheme is the industry standard for simplified models. It partitions water into three classes: Water Vapor (q_v), Cloud Water (q_c, tiny droplets that move with air), and Rain Water (q_r, large drops that fall).

For the user's "decay" problem, a simplified "Warm Rain" subset is sufficient:

- Saturation Adjustment: If $q_v > q_{\text{sat}}$, condense excess to q_c and heat the air.
- Autoconversion (Optional): If $q_c >$ threshold, convert to q_r.
- Sedimentation (Optional): q_r falls at terminal velocity.

Python Implementation Strategy:

```python
def saturation_adjustment(self):
    """
    Condenses excess water vapor into cloud water and releases
       latent heat.
    """
    L_v = 2.5e6        # Latent heat of vaporization (J/kg)
    c_p = 1004.0       # Specific heat of air (J/kg/K)
    Rv = 461.5         # Gas constant for water vapor

    # 1. Calculate Saturation Vapor Pressure (Tetens Formula)
    # T must be in Celsius for this specific formula
    T_degC = self.T - 273.15
    es = 6.112 * xp.exp((17.67 * T_degC) / (T_degC + 243.5)) *
       100.0 # Pascal

    # 2. Calculate Saturation Specific Humidity (q_sat)
    # p is in Pascal. epsilon = Rd/Rv = 0.622
    q_sat = 0.622 * es / (self.p - 0.378 * es)

    # 3. Determine Supersaturation
    # If qv > q_sat, we have condensation
    excess_q = self.qv - q_sat
    mask = excess_q > 0

    # 4. Apply Adjustment (Isobaric)
    # The amount condensed dq is approximated by:
    # dq = (qv - q_sat) / (1 + (L^2 * q_sat) / (cp * Rv * T^2))
    # This denominator is the "adjustment factor" accounting for
       the fact that
    # heating the air raises q_sat, limiting condensation.

    factor = 1.0 + (L_v**2 * q_sat) / (c_p * Rv * self.T**2)
    condensation = (excess_q / factor) * mask

    # Update Fields
    self.qv -= condensation
    self.qc += condensation  # Add to cloud water
    self.T += condensation * (L_v / c_p) # Latent heating
```

Integration: This function must be called inside the time step loop, specifically after the advection of T and q_v but before the buoyancy calculation. This ensures the buoyancy term "sees" the latent heat generated in that step.

4.3 Phase 3: Surface Fluxes (The Fuel Line)

The final component to prevent the "decay to < 10 knots" is the connection to the ocean. The boundary layer physics must be parameterized to represent the transfer of heat and moisture from the sea surface.

Bulk Aerodynamic Formula: Fluxes are modeled as proportional to the wind speed and the thermodynamic disequilibrium.

Table 1: Bulk Aerodynamic Flux Formulations

Flux Type	Formula	Components
Sensible Heat (H)	$\rho C_H \lvert\mathbf{u}\rvert (T_{\text{sfc}} - T_{\text{air}})$	Temperature difference
Latent Heat (E)	$\rho C_E \lvert\mathbf{u}\rvert (q^*_{\text{sat}} - q_{\text{air}})$	Moisture difference
Momentum (τ)	$\rho C_D \lvert\mathbf{u}\rvert \mathbf{u}$	Wind magnitude

Python Implementation:

```
def apply_surface_boundary_conditions(self, dt):
    """
    Applies WISHE fluxes to the lowest model layer.
    """
    # Coefficients (simplified constant values)
    Cd = 1.5e-3  # Drag coefficient
    Ch = 1.5e-3  # Heat exchange coefficient
    Ce = 1.5e-3  # Moisture exchange coefficient

    # Sea Surface State
    SST = 300.0  # 27 C (approx 80 F) - Fixed warm ocean

    # 1. Calculate Surface Wind Speed
    # Assuming lowest model level is index 0
    u_sfc = self.u[0, :, :] # Horizontal U
    v_sfc = self.v[0, :, :] # Horizontal V
    wspd = xp.sqrt(u_sfc**2 + v_sfc**2)

    # Add a "gustiness" factor to allow fluxes even at zero mean
       wind
    # This helps startup.
    wspd = xp.maximum(wspd, 1.0)

    # 2. Calculate Saturation at SST
    T_C_sst = SST - 273.15
    es_sst = 6.112 * xp.exp((17.67 * T_C_sst) / (T_C_sst + 243.5)
       ) * 100.0
    # Pressure at surface (approximate or extrapolated)
    p_sfc = self.p[0, :, :]
    q_sat_sst = 0.622 * es_sst / (p_sfc - 0.378 * es_sst)

    # 3. Calculate Fluxes
    # Sensible Heat (Temperature)
    flux_t = Ch * wspd * (SST - self.T[0, :, :])

    # Latent Heat (Moisture)
    flux_q = Ce * wspd * (q_sat_sst - self.qv[0, :, :])

    # Momentum Drag (Friction)
    stress_u = -Cd * wspd * u_sfc
    stress_v = -Cd * wspd * v_sfc

    # 4. Apply Tendencies to Lowest Layer
    # Divide flux by layer thickness (dz) to get tendency per
       second
    dz = self.dz

    self.T[0, :, :] += flux_t * (dt / dz)
    self.qv[0, :, :] += flux_q * (dt / dz)
    self.u[0, :, :] += stress_u * (dt / dz)
    self.v[0, :, :] += stress_v * (dt / dz)
```

Insight: This code closes the feedback loop. As the storm organizes, wspd increases in the eyewall region. This drives flux_q higher, pumping massive amounts of vapor into the boundary layer. This vapor is advected inward, rises (due to Boussinesq), condenses (Saturation Adjustment), and releases heat. This heat lowers the pressure, which further increases wspd. Without this function, the storm is physically impossible to sustain.

5 Detailed Analysis of the Simulation Failure Mode

5.1 The "Decay to 10 Knots" Phenomenon

The user reports the storm starts at the Genesis point but decays to < 10 knots by frame 300k. This indicates that numerical dissipation and physical friction are dominating the energy budget.

In numerical models, especially those solving Euler or Navier-Stokes on a grid, there is inherent numerical viscosity. If the user is using an "inviscid" solver, the discretization error (e.g., upwind differencing) acts as an artificial viscosity. In a real hurricane, the energy input from latent heat release is massive—on the order of 10^{14} Watts (200 times the worldwide electrical generating capacity).

In the user's model:

- Energy Sink: Surface friction (explicit or numerical) removes momentum.
- Energy Source: Currently Zero (or limited to initial potential energy in the warm bubble).
- Result: $d(\mathrm{KE})/dt < 0$. The vortex spins down.

By implementing Phase 3 (Surface Fluxes), the model introduces a source term S such that $d(\mathrm{KE})/dt = S - D$. If $S > D$ (which occurs when the thermodynamic disequilibrium is high and the vortex is organized), the storm intensifies.

5.2 The "Moist Air Just Sits There" Phenomenon

This confirms the Hydrostatic Paradox in Incompressible Solvers. In a compressible atmosphere, warm air expands. The pressure at the surface drops because the column weighs less. High pressure surrounds the low, driving inflow. In an incompressible solver, "expansion" is forbidden ($\nabla \cdot \mathbf{u} = 0$). You cannot have local expansion $\nabla \cdot \mathbf{u} > 0$ unless it is balanced by outflow elsewhere. Therefore, a temperature anomaly in an incompressible code does nothing mechanically unless explicitly coded into the momentum equation as a body force (gravity) via the Boussinesq term.

6 Numerical Considerations for Python Models

6.1 Performance and Vectorization

The user mentions "frame 300k". If this refers to 300,000 time steps, Python performance is a critical bottleneck.

Vectorization: Loops over grid points `for k in range(nz): for j...` are prohibitively slow in Python. All implementations provided above use xp (NumPy/CuPy) operations which execute in C/CUDA. This is essential for frames to compute in milliseconds rather than seconds.

Memory Layout: Arrays should be contiguous in memory. If iterating over z (vertical), ensure the z axis is the last axis (for C-order/NumPy default) or first axis (if optimized for that access pattern). Inconsistent memory access can slow down the pressure solver by orders of magnitude.

6.2 The Pressure Solver Bottleneck

The most expensive part of any incompressible solver is the Pressure Poisson Equation (PPE):

$$\nabla^2 p = R \tag{12}$$

In Python, this is often solved using:

- **FFT-based solvers:** If the grid is uniform and periodic, `numpy.fft` or `scipy.fft` is $O(N \log N)$ and extremely fast. This is the recommended approach for a "toy model."
- **Iterative solvers:** `scipy.sparse.linalg.cg` (Conjugate Gradient) or Multigrid methods. These are slower but handle complex boundaries better. If the user's "decay" is extremely slow (computationally), they might be using an inefficient solver (e.g., Jacobi iteration in pure Python). Switching to an FFT-based solver (if boundaries allow) could speed up the "300k frames" significantly.

7 Conclusion and Recommendation

The user's diagnosis is astute: the model lacks a rising mechanism. However, the proposed fix (Option D: Warm Core Init) is merely a setup step, not a dynamic solution. A warm bubble in a standard incompressible solver is dynamically inert.

To achieve a sustaining, intensifying tropical cyclone, the user must implement a layered physics engine:

1. **Reject "Quick Fix" Option D as a standalone solution.** It creates potential energy but provides no mechanism to release it.
2. **Implement Option C (Boussinesq) immediately.** This is the mechanical linkage that converts thermal anomalies into vertical motion.
3. **Implement Saturation Adjustment.** This converts the moisture from Option D into the heat required to sustain the Boussinesq buoyancy against adiabatic cooling.
4. **Implement WISHE Surface Fluxes.** This connects the storm to the ocean, providing the infinite energy reservoir required to counteract friction and sustain the vortex over 300k frames.

By integrating these three components, the Python model will transition from a kinematic fluid demonstration to a simplified but physically valid meteorological model.

This comparison highlights that Option D alone solves the moisture capacity issue but fails to address the dynamic coupling and energy maintenance issues that are causing the simulation failure.

Citations

- Boussinesq Approximation definition and utility.
- Incompressible flow solvers and the role of buoyancy terms.
- WISHE mechanism and hurricane maintenance.
- Bulk aerodynamic flux formulas for surface exchange.
- Kessler microphysics and saturation adjustment implementation.
- Carnot cycle view of TCs and Potential Intensity.

Table 2: Comparison of Model States

Feature	Current Model	Proposed Model (Option D only)	Required Model (Option D + C + Fluxes)
Base Temp	15°C	25°C	25°C
Moisture Capacity	Low	High	High
Vertical Motion Source	Pressure Solver (Kinematic)	Pressure Solver (Kinematic)	Buoyancy (Dynamic) + Kinematic
Response to Warm Core	None (Static)	None (Static)	Updraft Generation
Convection	None	None	Self-sustaining (Latent Heat)
Maintenance	Decays (Friction)	Decays (Friction)	Maintained (WISHE Feedback)
Outcome	Spin down	Spin down (slower?)	Intensification / Steady State