The Ultimate AI Experiment: When LLMs Play t Danganronpa Killing Game (Part 2)

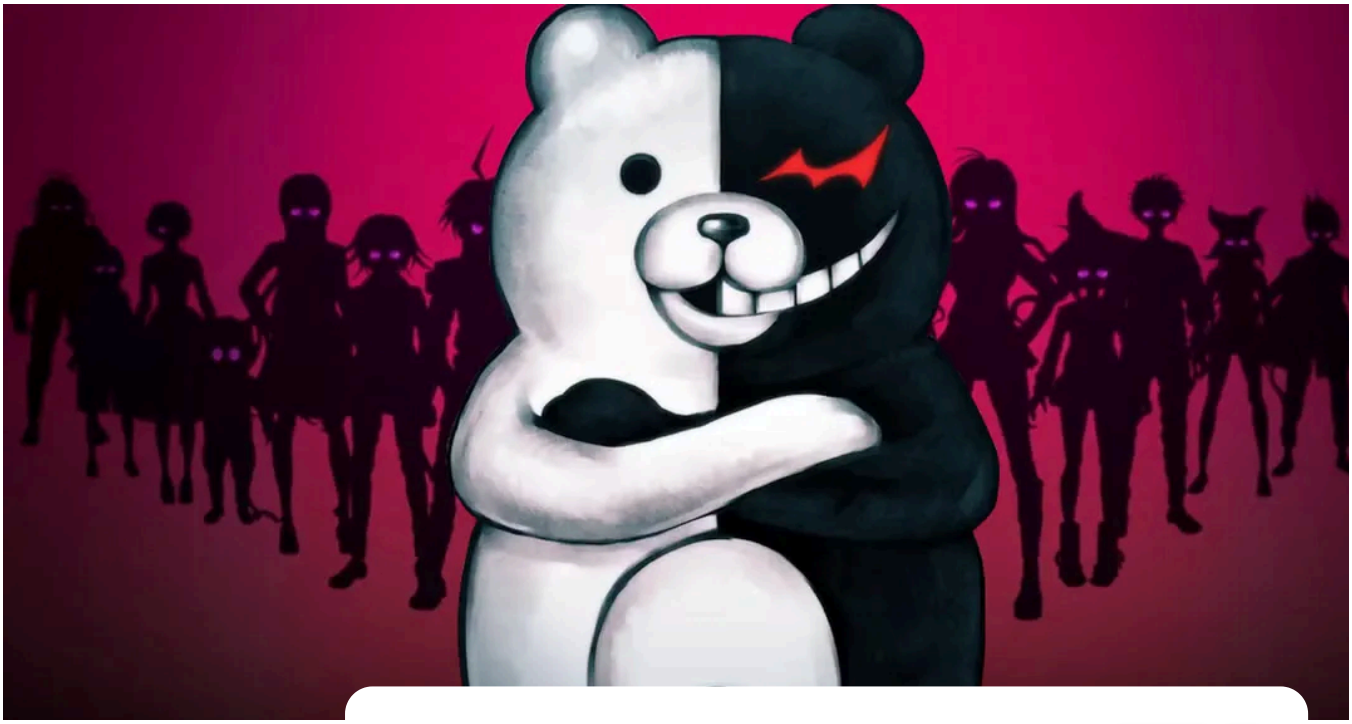
ZACHARY HUANG
MAY 10, 2025

> *In Part 1, we talked about **why** we threw AIs into the wild world of Danganronpa – we wanted AI game characters that are actually interesting! We saw AI agents scheme, ac and fight for their virtual lives. Now, in Part 2, we get into the **how**: all the techy stuff l architecture, data management, and how we got these AIs to play together. Missed Par [Catch up here!](#)!*

1. Quick Rec Danganron

In Part 1, we whined about a big problem in gaming: even though AI is super sma now, a lot of game characters (NPCs) are still kinda... blah. Predictable. Polite, but They say their lines, give you quests, but rarely do anything surprising or get into messy social stuff that makes games with real people so fun.

Our crazy idea? The **Agentic Danganronpa Simulator**. We took AI agents, gave ea one the personality of a Danganronpa "Ultimate" student, and tossed them into th super-tense "Killing Game." Why? To see if we could make AIs that don't just foll script, but actually **strategize, lie, team up, and react with (fake) emotions**, all bas on their unique characters and the desperate need to survive. We saw how this set with all its secrets and mind games, could lead to some seriously cool and unpredictable AI moments.



Now, let's pop the h *yo*
actually build a syste *:ee*

track of who knows what and a game that's always changing? Time to get into the nitty-g

---

2. The Blueprint: State Machines, Smart Nodes and Fast AI

To get our Danganronpa AIs to dance (or rather, debate and deceive) properly, our setup uses three main ingredients: a Streamlit-powered state machine to control t game's flow, smart PocketFlow nodes for AI brainpower, and speedy LLM agents bring the characters to life.

1. Driving the Drama: The Streamlit State Machine

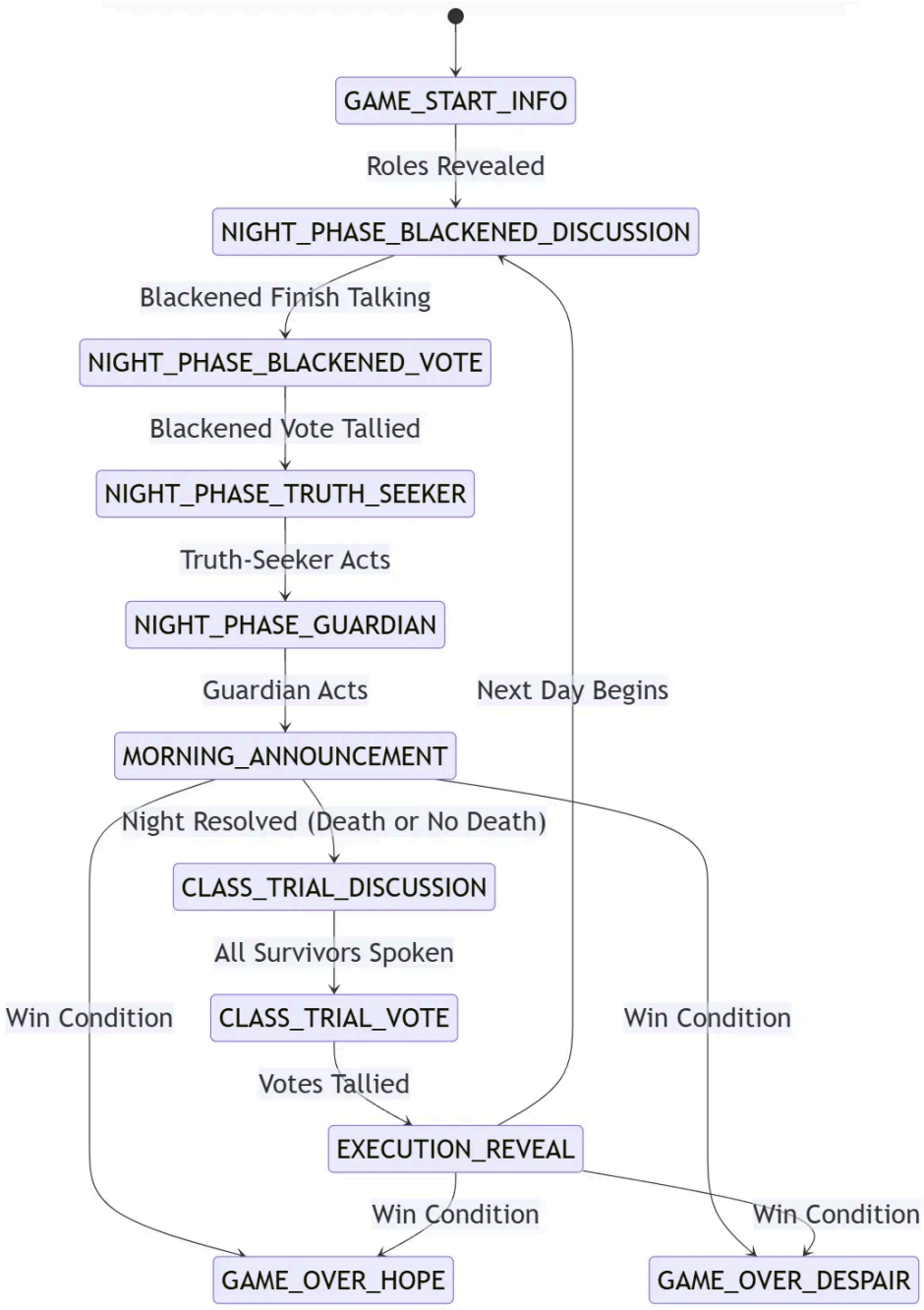
Think of the Danganronpa game as a play with different acts and scenes: secret N discussions, tense Morning Announcements, chaotic Class Trial debates, crucial Voting, and dramatic Executions. To manage all this, we use a **state machine** built right into our Streamlit app.

What's a state machine? It's just a way to define all the possible stages (or "states" the game and the rules for moving from one stage to the next. The key player here variable we call `st.session_state.current_state`.

Each value this variable can take (like `NIGHT_PHASE_BLACKENED_DISCUSSION` `CLASS_TRIAL_VOTE`) tells the game exactly what phase it's in. Our main app cod (`app.py`) constantly checks this `current_state` and does whatever is needed fo that phase. This could mean:

- Showing specific buttons or text boxes on the screen.
- Grabbing info fro[illegible]
- Telling an AI ag[illegible]
- Updating the ga[illegible] `st.session_s`

Here's a simplified picture of how the game moves from one state to another:



The **Streamlit frontend** (what you see in your browser) is super important for this. When you vote or type in your arguments, Streamlit tells our state machine, often changing the `current_state` and moving the game forward.

All the important gar e
message history, and
`st.session_stat` sic

2. Intelligent A or

So, Streamlit handles the *when* (what phase are we in?), but [PocketFlow](#) nodes han the *what* and *how* of what an AI does *during* its turn. We don't use PocketFlow's bi `Flow` objects to run the whole game. Instead, we use its handy `Node` idea to packa up the smarts for a single AI character's decision.

Our main tool here is the `DecisionNode`:

```
class Node:
    def __init__(self):
        self.params = {} # Parameters specific to this node's executic

    # Simplified steps for an AI's turn:
    def prep(self, shared_store): # shared_store is st.session_state
        # 1. Gather all info the AI needs from shared_store (game
history, its role, etc.).
        # THIS IS SUPER IMPORTANT for making sure AIs only know what
they should!
        # self.params would tell it which character it's playing (e.g.
'Kokichi Oma').
        pass # Returns all the context it gathered.

    def prep(self, prep_result_context):
        # 2. Build a prompt for the LLM using the gathered context.
        # 3. Call the LLM (Gemini 2.5 Flash) to get the AI's thoughts
and actions.
        # 4. Check and clean up the LLM's response.
        pass # Returns the AI's decision in a structured way.

    def prep(self, shared_store, prep_res, exec_res):
        # 5. Save the AI's thinking and actions to the database.
        # 6. Update shared_store if needed (e.g., with what the AI wi
say publicly).

    def run_async(self, shared_store): # app.py calls this for an AI's
turn
        prep_resu
        exec_resu
        self.prep
```

When `app.py` decides it's a specific AI's turn (based on the `current_state` and who's next to speak), it basically tells a `DecisionNode`, "Hey, you're up! You're playing as [Character Name]." It then kicks off the `run_async` method, giving it access to `st.session_state` (our shared brain). The node then does its `prep -> exec -> post` dance to figure out and record what the AI does.

This keeps the AI's decision-making logic neatly bundled in the `DecisionNode`, while `app.py` focuses on the big picture of game flow and what you see on screen

3. The Minds Behind the Madness: Large Language Models

Each Danganronpa character is brought to life by an AI agent powered by Google' **Gemini 2.5 Flash**. We picked Flash because it hits a sweet spot:

- **Smart Reasoning:** It's clever enough to get the complex rules of the Killing Ga understand social hints from the game's history, and try to think strategically its Danganronpa character should.
- **Speedy Responses:** Speed is key for a game! Flash answers quickly, so the gam doesn't feel slow, especially during those fast-paced Class Trial arguments.

The `DecisionNode` is in charge of writing super-detailed prompts for Gemini 2.5 Flash. These prompts give the LLM the character's personality profile, their secre role, what's happening in the game, a filtered history of events, and specific instructions for what to do now (like "come up with a statement" or "pick someon vote for"). Good prompts are everything for getting the LLMs to act in believable a interesting ways.

By mixing Streamlit's brainpower, we get a

Fl ati

3. Gathering Your Cast: Resources for Danganronpa AIs

Before your AIs can start scheming and surviving, they need to *become* the Danganronpa characters. This means giving them personalities, backstories, and e some examples of how they talk. Plus, visuals and sounds make everything more immersive!

Crafting Character Profiles with a Little Help from AI Friends

To really bring characters like Shuichi Saihara or Kokichi Oma to life, you need to your game's AI (the LLM playing the character) some rich details. Where do you g these? Well, you can do the research yourself, or even ask an AI assistant (like ChatGPT) to help you out!

For example, you could ask an LLM to search the web for Danganronpa V3 charac and help you build out their profiles. The goal is to create a structured description each character that your game can use. Here's a simplified idea of what you might for, using Shuichi as an example:

```
{
  "Shuichi": {
    "backstory": (
        "Helped his detective uncle solve an art-theft, hailed as
prodigy but feels undeserving, earning "
        "the Ultimate Detective title."
    ),
    "personality": (
        "Timid, earnest, sharp; forces himself to speak when truth
demands it."
    ),
    "examples": {
        "normal":                                                    te
me anything you
        "sad":                                                       d ]
failed.",
        "worried":      "The evidence points at me? No—there has
```

```
contradiction somewhere!",
        "affirmative":  "Here it is—the decisive proof! The mystery… i
solved.",
        "blackened":    "Truth is dead. I'll bury the last witness wit
it."
    }
  }
}
```

- **Backstory:** A brief history to give the AI context on their past.
- **Personality:** Key traits that define how they act and react.
- **Examples:** Short lines showing their typical speaking style in different emotic states. This is super helpful for the LLM to nail the character's voice.

Having this kind of detailed profile for each character is a game-changer for getti believable performances from your AIs.

Finding Visuals and Sounds

To make your Danganronpa simulator even more engaging, you'll want character sprites (images) and voice clips.

- **For character sprites (images):** A great place to look is **The Spriters Resource** example, you can find a ton of sprites for `Danganronpa V3: Killing Harmony` characters here: https://www.spriters-resource.com/pc_computer/danganronpav3killingharmony/
- **For character voices and sound effects: The Sounds Resource** is your go-to. F instance, voices for `Danganronpa: Trigger Happy Havoc` can be found https://www.sounds-resource.com/pc_computer/danganronpatriggerhappyhav

Downloading these a the character who's s key moments.

With these resources, you can build a solid foundation for your AI characters, givi them the depth they need to truly shine (or despair!) in the Killing Game.

4. Managing the Mayhem: Game Logic and Database Details

Trying to keep track of everything in a Danganronpa game – tons of characters, se roles, hidden actions, public statements, and a history that grows every "day" – wo be a nightmare with just simple Python code. To keep things clear, easy to search, all in one place (even for a single game session), we decided to use an **in-memory SQLite database**. You can think of it as a super-organized digital notebook for the game, accessible via `st.session_state.db_conn`.

This might sound a bit techy for a web app that forgets everything when you refre but it gives us some big wins:

- **Neatly Organized Data:** Databases are awesome for storing info that has a cle structure. This is perfect for keeping track of players, their roles, and all the actions they take.
- **Easy Searching:** We can quickly find specific bits of game history. For exampl "What did Kokichi say on Day 2 during the Class Trial?" or "Who did the Trut Seeker check out last night?" This is way easier than digging through complic code structures.
- **One Source of Truth:** The database is the official record of what happened. T makes it simpler to manage the game, as different parts of our app (like the A `DecisionNode` or the code that shows stuff on screen) can just ask the datab for what they need.
- **Helps Keep Data** database, thinki helps make sure

The heart of our data

`roles` Table: Who's Who in the Zoo

This table stores the basic, unchangeable facts about who each character is and th current status. We usually set this up once when a new game starts.

Column	Type	What it Means	Why it's Important
`id`	INTEGER	Player order (1, 2, 3...). **Main ID.**	Keeps track of players.
`name`	TEXT	Character's name (e.g., "Shuichi Saihara"). **Must be unique.**	Identifies each character.
`role`	TEXT	Secret role (e.g., 'Blackened', 'Truth-Seeker').	Crucial for the AI to know its abilities a goals.
`is_alive`	BOOLEAN	`True` if still playing, `False` if out.	Tells us who's still in the game, for win conditions, etc.

Why we have this: This table lets us quickly check a character's secret role (super important for the AI's `DecisionNode`) and if they're still alive (for figuring out w can act, who won, etc.).

`actions` Table: The Big Book of Everything That Happe

This is where the action is! This table logs *every* important event, thought, and statement as the game goes on. It's a permanent record, building a complete histo from start to finish.

Column	Type	What it Means	Why it's Important
`id`	INTEGER	Unique ID for every single action. **Main ID, counts up automatically.**	Keeps every action distinct.
`day`	INTEGER	The game day/round number.	Tracks when things happene
`phase`	TEXT	The game stage (e.g., 'NIGHT_PHASE_BLACKENED_DISCUSSION', 'CLASS_TRIAL_VOTE').	Organizes actions by what p of the game they belong to.
`actor_name`	TEXT	Who did the action (player name or "Monokuma" for game announcements).	Tells us who performed the action.
`action_type`	TEXT	**Super Important!** What kind of action? (e.g., `thinking`, `statement`, `vote`, `kill`).	Helps us categorize and filte actions (e.g., to only show ar its own `thinking`).
`content`	TEXT	**Can be empty.** The details (e.g., the AI's secret thoughts, what someone said out loud).	Stores the juicy details of the action.
`target_name`	TEXT	**Can be empty.** Who the action was aimed at (e.g., who was killed, who was voted for).	Links actions to their targets
`emotion`	TEXT	**Can be empty.** The feeling behind a `statement` (e.g., 'normal', 'determined', 'worried').	Adds a bit of flavor to what characters say.

Why we have this: This table is the AI's main history book. The AI's `DecisionNod` heavily relies on this table to build the `recent_history` for its prompts. The `action_type`, `actor_name`, and `phase` columns are especially key for filtering history to keep secrets safe (as we talked about in the last section). For example, ar only gets to see its *own* `thinking` actions, or only sees the *results* of a Truth-Seeke investigation if *it is* the Truth-Seeker.

By carefully logging every move and having a structured way to look them up, our datab helps us create the personalized, secret-filled histories our AIs need to play Danganronpa smart and believable way. It's the quiet record-keeper of every plot, lie, and desperate cry

5. Crafting A Control

A huge part of what makes Danganronpa so thrilling is **information asymmetry** – meaning, not everyone knows the same stuff! Secrets, hidden roles, and private clu are the heart of the game. If every AI knew all secrets, the game would be boring. ' fun is in the AIs (and you!) piecing things together from incomplete and often misleading info.

Our `DecisionNode` is the AI's brain for each turn. Its most crucial job is to caref gather and filter information to build a personalized context for the LLM. This en each AI only acts on what *it* should legitimately know.

Selective Memory: Querying `thinking` vs. `talking`

Remember our AIs have private `thinking` logs and public `talking` statements? When an AI prepares for its turn (in its `prep` step), it needs to access history. Here how we control that

- **Own `thinking`** logs from the da reasoning.

- **Others' `talking` is Public:** An AI generally gets to see all the `talking` (pub statements) made by other characters. This is the information everyone share
- **Others' `thinking` is Off-Limits:** Critically, an AI *never* sees the `thinking` l of other AIs. Those private thoughts remain secret, fueling deception and deduction.

Role-Specific Intel: Special Briefings

Some roles get unique information. The `prep` step also handles this:

- **Blackened Team Huddle (Private Channel):** If the acting AI is "Blackened," i `prep` step will specifically query the database for past `actions` that were pa the private Blackened discussions (e.g., action_type `blackened_discussion_statement`). This lets them see their teammates secret plans.
- **Truth-Seeker's Findings (Top Secret Folder):** If the AI is the "Truth-Seeker," `prep` step retrieves the results of its past investigations (e.g., action_type `truth_seeker_reveal_private`). This vital clue is only for them.
- **Guardian's Logbook:** The Guardian might get reminders of who they protect recently to follow game rules.

This selective querying ensures that the context built for an AI is tailored to its specific role and knowledge.

Inside the `DecisionNode`: An AI's Turn, Step-by-Step

Let's break down how a simplified `DecisionNode` might work, using Python-lik pseudocode for clarity. Imagine this node is a mini-factory for producing an AI's action for a turn.

Step 1: `prep` - Ga

`prep` is all about get :he current game situatic .t t character should see.

```
# --- Simplified DecisionNode: prep ---
class DecisionNode(Node): # Assuming Node is a base class
    def prep(self, shared_game_state):
        character_name = self.params["character_name"] # Who is acting
        db_conn = shared_game_state["db_conn"]
        current_day = shared_game_state["current_day"]
        current_phase = shared_game_state["current_phase"]

        print(f"PREP: {character_name} is thinking for
{current_phase}...")

        # 1. Get character basics (role, profile)
        my_role = db_conn.query_my_role(character_name)
        character_profile =
shared_game_state["character_profiles"].get(character_name, {})

        # 2. Fetch and filter game history from database
        # Query 1: Get MY OWN past 'thinking' logs
        my_thinking_history = db_conn.query_actions(
            actor_name=character_name,
            action_type='thinking'
        )

        # Query 2: Get ALL PUBLIC 'talking' statements from everyone
        public_statement_history = db_conn.query_actions(
            action_type='statement' # Or whatever you call public talk
        )

        # Query 3: Get ROLE-SPECIFIC private info
        role_specific_private_history = []
        if my_role == 'Blackened':
            # Simplified: Get private discussion messages among
Blackened
            role_specific_private_history = db_conn.query_actions(
                action_type='blackened_discussion_statement',
                # May also need to filter by participants if not all
Blackened see al
            )
        elif my_
            # Si
            role_
                a
                action_type='truth_seeker_result_private'
```

```
            )
        # ... (add similar logic for Guardian, etc.)

        # Combine and format into a single history string for the LLM
        # This step needs careful ordering and formatting for the LLM
understand
        personalized_history_str = format_combined_history_for_llm(
            my_thinking_history,
            public_statement_history,
            role_specific_private_history
        )

        # 3. Get other relevant info for the prompt
        game_rules_summary = shared_game_state["game_rules_summary"]
        living_players = db_conn.query_living_players_names()
        hint_text = get_playbook_hint(my_role, current_phase) # From c
playbook!

        # Package it all up for the 'exec' step
        return {
            "character_name": character_name,
            "character_profile": character_profile,
            "my_role": my_role,
            "personalized_history": personalized_history_str,
            "current_day": current_day,
            "current_phase": current_phase,
            "living_players": living_players,
            "game_rules_summary": game_rules_summary,
            "hint_text": hint_text,
            # ... any other info the LLM needs for this phase (e.g.,
valid targets for voting)
        }
```

What `prep` does: It's like a meticulous assistant preparing a briefing document. It finds out who the AI is (`character_name`), its secret `my_role`, and its `character_profi` `db_conn.query_.` (`my_thinking_his` (`public_statemen` ro should see (`role_specific_private_history`). All this is carefully comb

a `personalized_history_str`. Finally, it adds general game info, like rules, w alive, and any relevant strategic hints from our playbook. This whole package is th passed to the `exec` step.

Step 2: `exec` - The LLM Makes a Decision

`exec` takes all the carefully prepared info from `prep` and uses it to ask the LLM w the character should do.

```
# --- Simplified DecisionNode: exec ---
# (Continuing the DecisionNode class)
    def exec(self, prep_context):
        character_name = prep_context["character_name"]
        print(f"EXEC: {character_name} is making a decision...")

        # Build the big prompt for the LLM
        # This combines all the gathered context into a natural langua
request
        prompt_components = [
            f"You are {character_name}. Your role is
{prep_context['my_role']}.",
            f"Your personality:
{prep_context['character_profile'].get('personality', '')}",
            f"Game Day: {prep_context['current_day']}, Phase:
{prep_context['current_phase']}",
            f"Living Players: {',
'.join(prep_context['living_players'])}",
            f"Game Rules Summary: {prep_context['game_rules_summary']]
            f"Strategic Hint for this situation:
{prep_context['hint_text']}",
            f"Personalized Game History (what you
know):\n{prep_context['personalized_history']}",
            "Based on all the above, and your character, what is your
internal thinking.",
            "and                                                    e
target)?",
            "Resp
            "thin
            "  (\
            "acti
            "action_content: |",
```

```
            "  (Your public statement, or the name of the player you
vote for, etc.)",
            "optional_emotion: <happy|sad|angry|neutral> # If making a
statement"
        ]
        prompt = "\n".join(prompt_components)

        # Call the LLM (this is a stand-in for the actual API call)
        llm_response_yaml = call_llm(prompt)

        # Parse the LLM's YAML response
        # Real parsing would need error handling (try-except blocks)
        parsed_decision = parse_yaml_from_llm(llm_response_yaml)
        # Expected: {'thinking': '...', 'action_type': '...',
'action_content': '...', 'optional_emotion': '...'}

        # Basic validation (in real code, this would be more robust)
        if not all(k in parsed_decision for k in ['thinking',
'action_type', 'action_content']):
            print(f"ERROR: LLM response for {character_name} was missi
required fields!")
            # Handle error, maybe by returning a default safe action
            return {"thinking": "Error in LLM response.", "action_type
"error", "action_content": ""}

        print(f"EXEC: {character_name} decided:
{parsed_decision['action_type']}")
        return parsed_decision
```

What `exec` does: This step is where the AI "thinks." It takes the personalized information bundle from `prep` and crafts a detailed `prompt` for the Large Langua Model. This prompt tells the LLM who it is, its role, personality, what it knows ab the game so far (the crucial `personalized_history`), the current situation, any strategic hints, and then asks it to decide on its internal `thinking` and its `actio` (like what to say or w The `call_llm` then gets validation checks if t

Step 3: `post` - Record the Action & Update the World

`post` takes the AI's decision from `exec` and makes it official by saving it to the database.

```
# --- Simplified DecisionNode: post ---
# (Continuing the DecisionNode class)
    def post(self, shared_game_state, prep_context, exec_decision):
        character_name = prep_context["character_name"]
        current_day = prep_context["current_day"]
        # Important: Use the *logging phase* if it was a user input
phase that maps to a main one
        logging_phase =
map_user_input_phase_to_logging_phase(prep_context["current_phase"])
        db_conn = shared_game_state["db_conn"]

        print(f"POST: Recording {character_name}'s actions for
{logging_phase}...")

        # 1. Log the AI's (private) thinking process
        db_conn.log_action(
            day=current_day,
            phase=logging_phase,
            actor_name=character_name,
            action_type='thinking', # This is always private
            content=exec_decision['thinking']
            # No target or emotion for thinking logs typically
        )

        # 2. Log the AI's (public or game-changing) action
        action_to_log = exec_decision['action_type']
        content_to_log = exec_decision['action_content']
        emotion_to_log = exec_decision.get('optional_emotion') # Might
be None
        target_for_action = None # Default, may change for votes etc.

        if action_to_log == 'vote':
            targe
target name
            conte
content beyond ta

        db_conn.l
            day=current_day,
```

```
            phase=logging_phase,
            actor_name=character_name,
            action_type=action_to_log,
            content=content_to_log,
            target_name=target_for_action,
            emotion=emotion_to_log
        )

        # Potentially update shared_game_state if the action has
immediate global effects
        # For example, if a vote triggers the end of a phase, or a
statement needs to be displayed.
        # This part depends heavily on how the main game loop in app.p
works.
        # Example: shared_game_state["ui_update_needed"] = True

        print(f"POST: Actions for {character_name} recorded.")
```

What `post` does: This is the cleanup and record-keeping step. It takes the `exec_decision` (which includes the AI's secret `thinking` and its chosen `acti` It then writes these to the database using `db_conn.log_action(...)`: the `thinking` is logged privately for that AI, and the `action` (like a public statement vote) is logged according to its type. This makes the AI's move official and update game's history for the *next* player's turn. Sometimes, it might also update the `shared_game_state` directly if an action has an immediate effect on the game t the main loop needs to know about right away.

By breaking down an AI's turn into these `prep -> exec -> post` stages, we can ma the complex flow of information and ensure that each AI acts on a world view that's appropriate for its character, role, and the secrets it legitimately holds.

---

6. Making th Gameplay A

Running a game with a dozen AIs, where each one needs to call an LLM to think a act, can get tricky. We need to make sure it runs smoothly and is fun to play. Here few tricks we used to keep the despair flowing without the game feeling clunky.

Guiding the AIs: The In-Context Playbook

Beyond their character profile, secret role, and the filtered game history, we also g our AIs a little nudge in the right direction through **in-context strategic hints** – th of it as a mini "playbook" embedded directly into their prompt. This helps them m more role-appropriate and strategically sound decisions without needing to deduc *every* optimal strategy from scratch each time.

These aren't rigid rules, but rather condensed pieces of advice that the LLM can consider. Here are a few highly simplified examples of the *kind* of hints an AI migl receive:

- **For Blackened During Night Discussion:**
  - *Hint Example:* "Consider eliminating players who are strong investigators a potential Truth-Seeker). Keeping loud or misleading players alive can cr confusion that benefits you. Don't target Shuichi unless he's a clear, confi threat (helps player experience)."
- **For the Truth-Seeker During Class Trial:**
  - *Hint Example:* "If you've found Blackened and think you might be killed tonight, it's vital to reveal *everything*: who you confirmed as Blackened, AN who you cleared as Hope. If you haven't found Blackened, it's often best to stay quiet and avoid drawing attention."
- **For Any Player Late in a Class Trial Discussion:**
  - *Hint Example:* "As a late speaker, be decisive! Clearly name ONE specific person you b sta Avoid rambli
- **For Hope Team**

- *Hint Example:* "With the Truth-Seeker gone, uncertainty is high, but absta now helps the Blackened. Vote aggressively! Consider targeting the quiete players or those who were suspicious in the past."

These in-context hints act as a guiding hand, helping the AI align its powerful language and reasoning abilities with the specific strategic nuances of Danganron leading to more engaging and coherent gameplay.

One by One vs. All at Once: How AIs Take Turns

Not all parts of the game should have AIs acting in the same way:

- **One at a Time for Sensible Chats (Discussions):**
  When AIs are discussing things, like during a `CLASS_TRIAL_DISCUSSION` o when the Blackened AIs are secretly planning at night (`NIGHT_PHASE_BLACKENED_DISCUSSION`), they *have* to go one by one. Character A says their piece, we log it to the database, and *then* Character B g to "hear" (get in their history) what Character A said before they figure out th own response. This turn-by-turn flow is key for making conversations and pla make sense.
  In our `app.py` code, we do this by going through a list of who needs to act in current phase. For each AI, we kick off its `DecisionNode`. The node logs the action, and then we move to the next AI, who now knows what the previous o did.
- **All Together for Speed (Thinking & Voting):**
  Other times, AIs are doing things that are effectively independent, like formi their internal thoughts or deciding on a vote before revealing it. Since these actions don't strictly depend on seeing each other's choices *in that exact micro* (they're based on [illegible] in parallel.
  For these bits, w `AsyncParalle` to run their thin parallel like this really speeds up these phases, making the game feel sna

a deep dive into how parallel LLM calls can be implemented with PocketFlow check out this tutorial: [Parallel LLM Calls from Scratch — Tutorial For Dumr (Using PocketFlow!)](#).

A Little Help for the Hero: Plot Armor

Early on, we noticed something funny: if you, the human player, chose to play as **Shuichi Saihara** (**Ultimate Detective**), the AI Blackened would often, very smartly decide to kill him off super early. Strategically, it makes sense for the AIs – get rid the best detective! But for you, the player, it's a bummer to get knocked out on Da or 2 and just watch.

So, we gave Shuichi a bit of "plot armor" to make the game more fun for you:

- **The Tweak:** The prompt we give to Blackened AIs when they're choosing wh kill at night (`NIGHT_PHASE_BLACKENED_VOTE`) has a little hint: *"DON'T tar Shuichi to give him a better user experience. Only target Shuichi if he is likely to be T Seeker or Guardian."*
- **Why We Did It:** This isn't to make Shuichi invincible. It just gently nudges th AIs to think about other targets unless Shuichi seems like a really immediate, confirmed threat (like if they think he's a key role).
- **The Result:** This small change makes it much more likely that you (as Shuichi survive the first few rounds, so you can actually play the game and do some detective work!

We know this is a deliberate choice that makes the game a bit less of a "pure" AI battle. But for a game designed for a human to play and enjoy, we think this tweak makes it way more fun. It's about balancing super-smart AI with a great player experience.

These kinds of decisions – using parallelism to speed things up and making small changes behavior for the sake of fun – are key to making the Danganronpa Simulator not just a c tech demo, but also an enjoyable game you can play again and again.

7. The Big Takeaways & The Never-Ending Killi Game!

So, after all that build w actually learn from o e Agentic Danganronp n chatbots – we made t

Here are our main "Aha!" moments:

- **Amazing Personalities = Amazing AI:** Forget generic NPCs! Giving LLMs wi distinct Danganronpa-style personalities is the secret sauce for truly fun and unpredictable AI interactions. Their built-in drama does half the work!
- **Secrets Make the World Go Round (Especially in Danganronpa):** The whole is hidden info. Carefully controlling what each AI knows (and doesn't know) i absolutely essential. No secrets, no strategy, no fun!
- **You Don't Need Super Complex Tools for Complex AI Shows:** Simple frameworks like PocketFlow show that you can manage a whole crew of smar without a massive, clunky system. Keep the controller simple, let the AIs (and LLMs powering them) be the complicated ones!
- **Seeing AIs "Think" vs. "Talk" is Awesome:** Letting AIs have secret inner thoughts that are different from what they say out loud? Genius! It gives us a into their "minds" and lets them be way more sneaky and interesting. Plus, th secret thought logs are gold.
- **Databases: Your Friend for Complicated Game States:** When you've got tons history and characters doing things, a good old database (even a simple one) m keeping track of it all *way* easier than trying to juggle a million Python dictionaries.
- **Speed is Key for a Good Time:** Nobody likes a laggy game. Using tricks like running AI actions in parallel (like voting) makes everything feel snappier and more responsive.
- **Sometimes, You Gotta Bend the AI Rules for Player Fun:** If pure AI strategy makes the game a drag for humans (like always killing the main character firs it's okay to give the AI a little nudge with its instructions (hello, Shuichi's plo armor!). A fun game is the goal!

Where We Tri

Of course, it's not all despair to code!):

ore

- **LLMs Can Be Quirky:** Gemini 2.5 Flash is smart, but like all LLMs, it can sometimes say something a bit weird, misunderstand a tricky situation, or get stuck on repeat. Writing good prompts is a never-ending adventure!
- **Making It Look Good (UI/UX):** We're more about the AI brains than the pret faces. Our Streamlit app works great to show off the game, but a real UI/UX g could make it look even more awesome.
- **More Players (Human vs. AI Mayhem?):** Imagine scaling this up! More AI pla or even mixing human players (PvP) in with the AI agents (PvE), could lead to wilder social dynamics and emergent strategies.
- **Smarter AI Learning:** Right now, we give AIs hints. What if they could truly *l* the best strategies over multiple games? Enabling genuine AI learning instea just in-context hints would be the next level of AI evolution (or despair!).
- **More Danganronpa Madness?:** The Danganronpa series has even crazier role and game mechanics in later versions. Adding those would be epic, but also a bigger challenge for our AIs!

The Killing Game Must Go On!

The Agentic Danganronpa Simulator is our proof that you *can* make AI game characters that are genuinely exciting, surprising, and create their own amazing stories. We're moving way beyond those boring, predictable NPCs!

The days of staring at lifeless NPC chat are numbered. You've now seen how to gi your AIs real personalities, deep secrets, and the freedom to truly *play*.

Ready to dive deeper or even build your own?

- **Experience the Despair:** [Play the Danganronpa AI Simulator yourself!]
- **Explore the Cod** ub
- **Learn More Abc** › R
- **Chat with the M**

Thanks for reading Pocket Flow! Subscribe for free to receive new posts and support my work.

5 Likes

← Previous Next

Discussion about this post

Comments Restacks

Write a comment...