Analysis of Profiling Results

1. **The Biggest Time Consumer: The Game Loop's Frame Rate**

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
13775	98.166	0.007	98.166	0.007	{method 'tick' of 'pygame.time.Clock' objects}

The single most time-consuming function is **pygame.time.Clock.tick**, which takes **98.166 seconds** (over 54% of the total execution time).

Remedial Action: Lower the FPS to a more reasonable value like 30 or 60

2. **The Next Biggest Time Consumer: AI Network Latency**

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
150	58.885	0.393	58.885	0.393	{method 'read' of '_ssl._SSLSocket' objects}

The **get_ai_move** function and its underlying network calls (through **openai**, **httpx**, and **ssl**) are the next major bottleneck, consuming about **60 seconds** in total. The function {method 'read' of '_ssl._SSLSocket' objects} alone took **58.8 seconds**.

Reason: This time is spent waiting for a response from the DeepSeek API over the internet. Program is paused, waiting for an external resource, that cannot be optimized

Remedial Action: Notifying human player that AI opponent is thinking. Implement and render **thinking_text** as "AI is thinking..."
Also, will consider using function caching in the ai.py module, as Python's standard library provides a simple way to do this with the functools.lru_cache decorator.
A function decorated with @lru_cache will automatically store its recent results in a dictionary-like cache

I have also chosen to profile the AI's turn because it is the only part of the game logic that introduces a noticeable delay and is therefore a candidate for analysis and improvement. Isolating this code block by encapsulating it with the **import cProfile, pstats,** calling **profiler = cProfile.Profile()** and **profiler.enable()** before the call **get_ai_move(board.gameplay),** and stopped profiling after the AI move with **profiler.disable(),** subsequently printing the stats for just that turn with **stats = pstats.Stats(profiler).sort_stats('cumulative')** and **stats.print_stats(10)**

3. **Inefficient Drawing in the Game Loop**

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
13775	4.396	0.000	4.396	0.000	{built-in method pygame.display.update}
336589	2.406	0.000	2.406	0.000	{method 'blit' of 'pygame.surface.Surface'
336589	2.401	0.000	2.401	0.000	{method 'render' of 'pygame.font.Font' objects}
578550	1.642	0.000	1.642	0.000	{built-in method pygame.draw.circle}

The functions draw_matrix_info, board.draw_board, pygame.font.Font.render, and pygame.Surface.blit are called thousands of times and collectively consume several seconds.

Reason: draw_matrix_info is called in every single frame of the game loop. It re-renders the text for all three tracking matrices from scratch, even when the data hasn't changed. This is inefficient

Remedial Action: Refactor drawing logic to only redraw static elements like the board and matrices when the game state actually changes, not on every single frame.

a. Implement a flag in class Board's constructor (in module board.py) to indicate if the board needs to be redrawn. **Set flag to true** at object instantiation to ensure the initial rendering of the game board.
b. **Set flag to true** in **drop_piece** method as it changes board when it is called.
c. **Test flag** before calling **draw_board** and **draw_matrix_info** in main.py and **set to false** ensuring subsequent iterations of the 'while run' loop only enter this block if the **drop_piece** method is called and changes the flag. This ensures the 'while run' loop in main.py draws the board and matrix information only when the board has changed