```
import asyncio
import logging
import os
import time
from abc import abstractmethod
from typing import List, Optional


class BaseLLM:
    """Abstract Language Model that defines the interface for all language models

    Args:
        model_name (Optional[str], optional): _description_. Defaults to None.
        max_tokens (Optional[int], optional): _description_. Defaults to None.
        max_length (Optional[int], optional): _description_. Defaults to None.
        temperature (Optional[float], optional): _description_. Defaults to None.
        top_k (Optional[float], optional): _description_. Defaults to None.
        top_p (Optional[float], optional): _description_. Defaults to None.
        system_prompt (Optional[str], optional): _description_. Defaults to None.
        beam_width (Optional[int], optional): _description_. Defaults to None.
        num_return_sequences (Optional[int], optional): _description_. Defaults to None.
        seed (Optional[int], optional): _description_. Defaults to None.
        frequency_penalty (Optional[float], optional): _description_. Defaults to None.
        presence_penalty (Optional[float], optional): _description_. Defaults to None.
        stop_token (Optional[str], optional): _description_. Defaults to None.
        length_penalty (Optional[float], optional): _description_. Defaults to None.
```

role (Optional[str], optional): _description_. Defaults to None.

do_sample (Optional[bool], optional): _description_. Defaults to None.

early_stopping (Optional[bool], optional): _description_. Defaults to None.

num_beams (Optional[int], optional): _description_. Defaults to None.

repition_penalty (Optional[float], optional): _description_. Defaults to None.

pad_token_id (Optional[int], optional): _description_. Defaults to None.

eos_token_id (Optional[int], optional): _description_. Defaults to None.

bos_token_id (Optional[int], optional): _description_. Defaults to None.

device (Optional[str], optional): _description_. Defaults to None.

*args: _description_

**kwargs: _description_

"""

```
def __init__(
    self,
    model_id: Optional[str] = None,
    model_name: Optional[str] = None,
    max_tokens: Optional[int] = None,
    max_length: Optional[int] = None,
    temperature: Optional[float] = None,
    top_k: Optional[float] = None,
    top_p: Optional[float] = None,
    system_prompt: Optional[str] = None,
    beam_width: Optional[int] = None,
```

```
    num_return_sequences: Optional[int] = None,
    seed: Optional[int] = None,
    frequency_penalty: Optional[float] = None,
    presence_penalty: Optional[float] = None,
    stop_token: Optional[str] = None,
    length_penalty: Optional[float] = None,
    role: Optional[str] = None,
    do_sample: Optional[bool] = None,
    early_stopping: Optional[bool] = None,
    num_beams: Optional[int] = None,
    repition_penalty: Optional[float] = None,
    pad_token_id: Optional[int] = None,
    eos_token_id: Optional[int] = None,
    bos_token_id: Optional[int] = None,
    device: Optional[str] = None,
    freq_penalty: Optional[float] = None,
    stop_token_id: Optional[int] = None,
    *args,
    **kwargs,
):
    super().__init__(*args, **kwargs)
    self.model_id = model_id
    self.model_name = model_name
    self.max_tokens = max_tokens
    self.temperature = temperature
    self.top_k = top_k
```

```
self.top_p = top_p
self.system_prompt = system_prompt
self.beam_width = beam_width
self.num_return_sequences = num_return_sequences
self.seed = seed
self.frequency_penalty = frequency_penalty
self.presence_penalty = presence_penalty
self.stop_token = stop_token
self.length_penalty = length_penalty
self.role = role
self.max_length = max_length
self.do_sample = do_sample
self.early_stopping = early_stopping
self.num_beams = num_beams
self.repition_penalty = repition_penalty
self.pad_token_id = pad_token_id
self.eos_token_id = eos_token_id
self.bos_token_id = bos_token_id
self.device = device
self.frequency_penalty = freq_penalty
self.stop_token_id = stop_token_id

# Attributes
self.history = ""
self.start_time = None
self.end_time = None
```

```
    self.history = []

@abstractmethod
def run(self, task: Optional[str] = None, *args, **kwargs) -> str:
    """generate text using language model"""

async def arun(self, task: Optional[str] = None, *args, **kwargs):
    """Asynchronous run

    Args:
        task (Optional[str], optional): _description_. Defaults to None.
    """
    loop = asyncio.get_event_loop()
    result = await loop.run_in_executor(None, self.run, task)
    return result

def batch_run(self, tasks: List[str], *args, **kwargs):
    """Batch run with language model

    Args:
        tasks (List[str]): _description_

    Returns:
        _type_: _description_
    """
    return [self.run(task) for task in tasks]
```

```
async def abatch_run(self, tasks: List[str], *args, **kwargs):
    """Asynchronous batch run with language model

    Args:
        tasks (List[str]): _description_

    Returns:
        _type_: _description_
    """
    return await asyncio.gather(
        *(self.arun(task) for task in tasks)
    )

def chat(self, task: str, history: str = "") -> str:
    """Chat with the model"""
    complete_task = (
        task + " | " + history
    )  # Delimiter for clarity
    return self.run(complete_task)

def __call__(self, task: str) -> str:
    """Call the model"""
    return self.run(task)

def _tokens_per_second(self) -> float:
```

```
    """Tokens per second"""
    elapsed_time = self.end_time - self.start_time
    if elapsed_time == 0:
        return float("inf")
    return self._num_tokens() / elapsed_time

# def _num_tokens(self, text: str) -> int:
# """Number of tokens"""
# tokenizer = self.tokenizer
# return count_tokens(text)

def _time_for_generation(self, task: str) -> float:
    """Time for Generation"""
    self.start_time = time.time()
    self.run(task)
    self.end_time = time.time()
    return self.end_time - self.start_time

def generate_summary(self, text: str) -> str:
    """Generate Summary"""

def set_temperature(self, value: float):
    """Set Temperature"""
    self.temperature = value

def set_max_tokens(self, value: int):
```

```
    """Set new max tokens"""
    self.max_tokens = value

def clear_history(self):
    """Clear history"""
    self.history = []

def enable_logging(self, log_file: str = "model.log"):
    """Initialize logging for the model."""
    logging.basicConfig(filename=log_file, level=logging.INFO)
    self.log_file = log_file

def log_event(self, message: str):
    """Log an event."""
    logging.info(
        f"{time.strftime('%Y-%m-%d %H:%M:%S')} - {message}"
    )

def save_checkpoint(self, checkpoint_dir: str = "checkpoints"):
    """Save the model state."""
    # This is a placeholder for actual checkpointing logic.
    if not os.path.exists(checkpoint_dir):
        os.makedirs(checkpoint_dir)
    checkpoint_path = os.path.join(
        checkpoint_dir,
        f'checkpoint_{time.strftime("%Y%m%d-%H%M%S")}.ckpt',
```

```
    )
    # Save model state to checkpoint_path
    self.log_event(f"Model checkpoint saved at {checkpoint_path}")

def load_checkpoint(self, checkpoint_path: str):
    """Load the model state from a checkpoint."""
    # This is a placeholder for actual loading logic.
    # Load model state from checkpoint_path
    self.log_event(f"Model state loaded from {checkpoint_path}")

def toggle_creative_mode(self, enable: bool):
    """Toggle creative mode for the model."""
    self.creative_mode = enable
    self.log_event(
        f"Creative mode {'enabled' if enable else 'disabled'}."
    )

def track_resource_utilization(self):
    """Track and report resource utilization."""
    # This is a placeholder for actual tracking logic.
    # Logic to track CPU, memory, etc.
    utilization_report = "Resource utilization report here"
    return utilization_report

def get_generation_time(self) -> float:
    """Get generation time"""
```

```
    if self.start_time and self.end_time:
        return self.end_time - self.start_time
    return 0

def set_max_length(self, max_length: int):
    """Set max length

    Args:
        max_length (int): _description_
    """
    self.max_length = max_length

def set_model_name(self, model_name: str):
    """Set model name

    Args:
        model_name (str): _description_
    """
    self.model_name = model_name

def set_frequency_penalty(self, frequency_penalty: float):
    """Set frequency penalty

    Args:
        frequency_penalty (float): _description_
    """
```

```
    self.frequency_penalty = frequency_penalty

def set_presence_penalty(self, presence_penalty: float):
    """Set presence penalty

    Args:
        presence_penalty (float): _description_
    """
    self.presence_penalty = presence_penalty

def set_stop_token(self, stop_token: str):
    """Set stop token

    Args:
        stop_token (str): _description_
    """
    self.stop_token = stop_token

def set_length_penalty(self, length_penalty: float):
    """Set length penalty

    Args:
        length_penalty (float): _description_
    """
    self.length_penalty = length_penalty
```

```
def set_role(self, role: str):
    """Set role

    Args:
        role (str): _description_
    """
    self.role = role

def set_top_k(self, top_k: int):
    """Set top k

    Args:
        top_k (int): _description_
    """
    self.top_k = top_k

def set_top_p(self, top_p: float):
    """Set top p

    Args:
        top_p (float): _description_
    """
    self.top_p = top_p

def set_num_beams(self, num_beams: int):
    """Set num beams
```

```
    Args:
        num_beams (int): _description_
    """
    self.num_beams = num_beams

def set_do_sample(self, do_sample: bool):
    """set do sample



    Args:
        do_sample (bool): _description_
    """
    self.do_sample = do_sample

def set_early_stopping(self, early_stopping: bool):
    """set early stopping

    Args:
        early_stopping (bool): _description_
    """
    self.early_stopping = early_stopping

def set_seed(self, seed: int):
    """Set seed
```

```
    Args:
        seed ([type]): [description]
    """
    self.seed = seed

def set_device(self, device: str):
    """Set device

    Args:
        device (str): _description_
    """
    self.device = device

def metrics(self) -> str:
    """
    Metrics

    Returns:
        str: _description_
    """
    _sec_to_first_token = self._sec_to_first_token()
    _tokens_per_second = self._tokens_per_second()
    _num_tokens = self._num_tokens(self.history)
    _time_for_generation = self._time_for_generation(self.history)

    return f"""
```

```
    SEC TO FIRST TOKEN: {_sec_to_first_token}
    TOKENS/SEC: {_tokens_per_second}
    TOKENS: {_num_tokens}
    Tokens/SEC: {_time_for_generation}
    """

def time_to_first_token(self, prompt: str) -> float:
    """Time to first token

    Args:
        prompt (str): _description_

    Returns:
        float: _description_
    """
    start_time = time.time()
    self.track_resource_utilization(
        prompt
    )  # assuming `generate` is a method that generates tokens
    first_token_time = time.time()
    return first_token_time - start_time

def generation_latency(self, prompt: str) -> float:
    """generation latency

    Args:
```

```
        prompt (str): _description_

    Returns:
        float: _description_
    """
    start_time = time.time()
    self.run(prompt)
    end_time = time.time()
    return end_time - start_time

def throughput(self, prompts: List[str]) -> float:
    """throughput

    Args:
        prompts (): _description_

    Returns:
        float: _description_
    """
    start_time = time.time()
    for prompt in prompts:
        self.run(prompt)
    end_time = time.time()
    return len(prompts) / (end_time - start_time)
```