Launcher

```
fn main() -> Result<(), LauncherError> {
    // Pattern match configuration
    let args: Args = Args::parse();

    // Filter events with LOG_LEVEL
    let env_filter =
        EnvFilter::try_from_env("LOG_LEVEL").unwrap_or_else(|_| EnvFilter::new("info"));

    if args.json_output {
        tracing_subscriber::fmt()
            .with_env_filter(env_filter)
            .json()
            .init();
    } else {
        tracing_subscriber::fmt()
            .with_env_filter(env_filter)
            .compact()
            .init();
    }

    if args.env {
        let env_runtime = env_runtime::Env::new();
        tracing::info!("{}", env_runtime);
    }

    tracing::info!("{:?}", args);

    // Validate args
    if args.max_input_length >= args.max_total_tokens {
        return Err(LauncherError::ArgumentValidation(
            "`max_input_length` must be < `max_total_tokens`".to_string(),
        ));
    }
    if args.max_input_length as u32 > args.max_batch_prefill_tokens {
        return Err(LauncherError::ArgumentValidation(format!(
            "`max_batch_prefill_tokens` must be >= `max_input_length`. Given: {} and {}",
            args.max_batch_prefill_tokens, args.max_input_length
        )));
    }

    if args.validation_workers == 0 {
        return Err(LauncherError::ArgumentValidation(
            "`validation_workers` must be > 0".to_string(),
        ));
```

```
}
if args.trust_remote_code {
    tracing::warn!(
        "`trust_remote_code` is set. Trusting that model `{}` do not contain malicious c
        args.model_id
    );
}

let num_shard = find_num_shards(args.sharded, args.num_shard)?;
if num_shard > 1 {
    tracing::info!("Sharding model on {num_shard} processes");
}

if let Some(ref max_batch_total_tokens) = args.max_batch_total_tokens {
    if args.max_batch_prefill_tokens > *max_batch_total_tokens {
        return Err(LauncherError::ArgumentValidation(format!(
            "`max_batch_prefill_tokens` must be <= `max_batch_total_tokens`. Given: {} a
            args.max_batch_prefill_tokens, max_batch_total_tokens
        )));
    }
    if args.max_total_tokens as u32 > *max_batch_total_tokens {
        return Err(LauncherError::ArgumentValidation(format!(
            "`max_total_tokens` must be <= `max_batch_total_tokens`. Given: {} and {}",
            args.max_total_tokens, max_batch_total_tokens
        )));
    }
}

if args.ngrok {
    if args.ngrok_authtoken.is_none() {
        return Err(LauncherError::ArgumentValidation(
            "`ngrok-authtoken` must be set when using ngrok tunneling".to_string(),
        ));
    }

    if args.ngrok_edge.is_none() {
        return Err(LauncherError::ArgumentValidation(
            "`ngrok-edge` must be set when using ngrok tunneling".to_string(),
        ));
    }
}

// Signal handler
let running = Arc::new(AtomicBool::new(true));
let r = running.clone();
ctrlc::set_handler(move || {
    r.store(false, Ordering::SeqCst);
})
.expect("Error setting Ctrl-C handler");

// Download and convert model weights
download_convert_model(&args, running.clone())?;

if !running.load(Ordering::SeqCst) {
    // Launcher was asked to stop
    return Ok(());
}

// Shared shutdown bool
let shutdown = Arc::new(AtomicBool::new(false));
// Shared shutdown channel
// When shutting down, the main thread will wait for all senders to be dropped
let (shutdown_sender, shutdown_receiver) = mpsc::channel();

// Shared channel to track shard status
let (status_sender, status_receiver) = mpsc::channel();

spawn_shards(
    num_shard,
    &args,
    shutdown.clone(),
    &shutdown_receiver,
    shutdown_sender,
    &status_receiver,
    status_sender,
    running.clone(),
)?;

// We might have received a termination signal
if !running.load(Ordering::SeqCst) {
    shutdown_shards(shutdown, &shutdown_receiver);
    return Ok(());
}

let mut webserver = spawn_webserver(num_shard, args, shutdown.clone(), &shutdown_receive
    .map_err(|err| {
        shutdown_shards(shutdown.clone(), &shutdown_receiver);
        err
    })?;

// Default exit code
let mut exit_code = Ok(());
```

```
    while running.load(Ordering::SeqCst) {
        if let Ok(ShardStatus::Failed(rank)) = status_receiver.try_recv() {
            tracing::error!("Shard {rank} crashed");
            exit_code = Err(LauncherError::ShardFailed);
            break;
        };

        match webserver.try_wait().unwrap() {
            Some(_) => {
                tracing::error!("Webserver Crashed");
                shutdown_shards(shutdown, &shutdown_receiver);
                return Err(LauncherError::WebserverFailed);
            }
            None => {
                sleep(Duration::from_millis(100));
            }
        };
    }

    // Graceful termination
    terminate("webserver", webserver, Duration::from_secs(90)).unwrap();
    shutdown_shards(shutdown, &shutdown_receiver);

    exit_code
}
```

Above is the main fn for the rust launcher. When we compile and run the program, we observe that the launcher starts first.

inside the launcher, it + define the number of shards based on command line argument
+ spawn a new server process to download the model weights + Spawn a certain number of shard managers given the command line arguments, where each thread starts a server process
+ Spawn a single webserver, aka the router, to manage the different threads. + Each server process has a unique rank that is assigned by the router via env variables.

Server (Python)

- construct local urls and urls for the other shards based on ranks for communication with grpc, the url looks like this /tmp/text-generation-server - {rank}
- register for the grpc service on the machine

- get_model method parse the input parameter, and fetch the corresponding model.

```
    try:
        model = get_model(
            model_id,
            revision,
            sharded,
            quantize,
            speculate,
            dtype,
            trust_remote_code,
        )
    except Exception:
        logger.exception("Error when initializing model")
        raise
```

- if looking for a specific model implementation, then fetch it (e.g. flash llama). Else, always fall back to the general case, CausalLM, which uses the Transformers library methods like fromPretrained to load the model and the tokenizer:

```
elif model_type == "llama" or model_type == "baichuan":
    if FLASH_ATTENTION:
        return FlashLlama(
            model_id,
            revision,
            quantize=quantize,
            dtype=dtype,
            trust_remote_code=trust_remote_code,
            use_medusa=use_medusa,
        )
    elif sharded:
        raise NotImplementedError(FLASH_ATT_ERROR_MESSAGE.format("Sharded Llama"))
    else:
        return CausalLM(
            model_id,
            revision,
            quantize=quantize,
            dtype=dtype,
            trust_remote_code=trust_remote_code,
        )
```

- inside the implementation, the flash version of llama uses accelerated implementation:

```
class FlashLlamaForCausalLM(torch.nn.Module):
    def __init__(self, config, weights):
        super().__init__()

        self.model = FlashLlamaModel(config, weights)
        self.lm_head = TensorParallelHead.load(
            config,
            prefix="lm_head",
            weights=weights,
        )
```

- It seems to be using pytorch distribute to do tensor parallel and quantization (no flash attention package involved, though it is written in the util). It is coordinating with other shards via the pytorch module.
- Inside each model implementation, it handles the generate and speculative decoding, nothing fancy in particular.

Router

The router (also referred to as the webserver) is actually coordinating different shards to do the Infer task. When a user submits a request, it is submitted to the router. THe router handles the logic for continuous batching as well.

First look at the main function of the router rust code: It does argument parsing, load tokenizer (only for validation, tokenization is done on the shard). Warm up the model.

```rust
// Instantiate sharded client from the master unix socket
let mut sharded_client = ShardedClient::connect_uds(master_shard_uds_path)
    .await
    .map_err(RouterError::Connection)?;
// Clear the cache; useful if the webserver rebooted
sharded_client
    .clear_cache(None)
    .await
    .map_err(RouterError::Cache)?;
// Get info from the shard
let shard_info = sharded_client.info().await.map_err(RouterError::Info)?;
```

this code is weird, I have yet to find out the meaning of a sharded client. Apparently onr router has a single sharded client. It seems to me that the client is responsible for communicating with different shards. I have not read the corresponding code.

Then the main function starts running the router code by server::run()

```rust
// Define base and health routes
let base_routes = Router::new()
    .route("/", post(compat_generate))
    .route("/", get(health))
```

```
        .route("/info", get(get_model_info))
        .route("/generate", post(generate))
        .route("/generate_stream", post(generate_stream))
        .route("/v1/chat/completions", post(chat_completions))
        .route("/tokenize", post(tokenize))
        .route("/health", get(health))
        .route("/ping", get(health))
        .route("/metrics", get(metrics));

    // Conditional AWS Sagemaker route
    let aws_sagemaker_route = if messages_api_enabled {
        Router::new().route("/invocations", post(chat_completions)) // Use 'chat_completions
    } else {
        Router::new().route("/invocations", post(compat_generate)) // Use 'compat_generate'
    };

    let compute_type =
        ComputeType(std::env::var("COMPUTE_TYPE").unwrap_or("gpu+optimized".to_string()));

    // Combine routes and layers
    let app = Router::new()
        .merge(swagger_ui)
        .merge(base_routes)
        .merge(aws_sagemaker_route)
        .layer(Extension(info))
        .layer(Extension(health_ext.clone()))
        .layer(Extension(compat_return_full_text))
        .layer(Extension(infer))
        .layer(Extension(compute_type))
        .layer(Extension(prom_handle.clone()))
        .layer(OtelAxumLayer::default())
        .layer(cors_layer);

    if ngrok {
        #[cfg(feature = "ngrok")]
        {
            use ngrok::config::TunnelBuilder;

            let _ = addr;

            let authtoken =
                ngrok_authtoken.expect("`ngrok-authtoken` must be set when using ngrok tunne

            let edge = ngrok_edge.expect("`ngrok-edge` must be set when using ngrok tunnelin

            let tunnel = ngrok::Session::builder()
```

```rust
            .authtoken(authtoken)
            .connect()
            .await
            .unwrap()
            .labeled_tunnel()
            .label("edge", edge);

        let listener = tunnel.listen().await.unwrap();

        // Run prom metrics and health locally too
        tokio::spawn(
            axum::Server::bind(&addr)
                .serve(
                    Router::new()
                        .route("/health", get(health))
                        .route("/metrics", get(metrics))
                        .layer(Extension(health_ext))
                        .layer(Extension(prom_handle))
                        .into_make_service(),
                )
                //Wait until all requests are finished to shut down
                .with_graceful_shutdown(shutdown_signal()),
        );

        // Run server
        axum::Server::builder(listener)
            .serve(app.into_make_service())
            //Wait until all requests are finished to shut down
            .with_graceful_shutdown(shutdown_signal())
            .await?;
    }
    #[cfg(not(feature = "ngrok"))]
    {
        let _ngrok_authtoken = ngrok_authtoken;
        let _ngrok_domain = ngrok_domain;
        let _ngrok_username = ngrok_username;
        let _ngrok_password = ngrok_password;

        panic!("`text-generation-router` was compiled without the `ngrok` feature");
    }
} else {
    // Run server
    axum::Server::bind(&addr)
        .serve(app.into_make_service())
        // Wait until all requests are finished to shut down
        .with_graceful_shutdown(shutdown_signal())
```

```
            .await?;
    }
    Ok(())
}
```

as shown in the code above, the most important part of the server run code is to define different routes (the router is essentially a web server to the client submitting their requests) and add middleware implementation to the axum rust server. It also binds the server to a certain address, and if ngrok is configured, it can also accept general internet requests, instead of local requests only.

```
/// Generate tokens
#[utoipa::path(
post,
tag = "Text Generation Inference",
path = "/generate",
request_body = GenerateRequest,
responses(
(status = 200, description = "Generated Text", body = GenerateResponse),
(status = 424, description = "Generation Error", body = ErrorResponse,
example = json ! ({"error": "Request failed during generation"})),
(status = 429, description = "Model is overloaded", body = ErrorResponse,
example = json ! ({"error": "Model is overloaded"})),
(status = 422, description = "Input validation error", body = ErrorResponse,
example = json ! ({"error": "Input validation error"})),
(status = 500, description = "Incomplete generation", body = ErrorResponse,
example = json ! ({"error": "Incomplete generation"})),
)
)]
#[instrument(
skip_all,
fields(
parameters = ? req.parameters,
total_time,
validation_time,
queue_time,
inference_time,
time_per_token,
seed,
)
)]
async fn generate(
    infer: Extension<Infer>,
    Extension(ComputeType(compute_type)): Extension<ComputeType>,
    Json(req): Json<GenerateRequest>,
) -> Result<(HeaderMap, Json<GenerateResponse>), (StatusCode, Json<ErrorResponse>)> {
    let span = tracing::Span::current();
```

```rust
let start_time = Instant::now();
metrics::increment_counter!("tgi_request_count");

tracing::debug!("Input: {}", req.inputs);

let compute_characters = req.inputs.chars().count();
let mut add_prompt = None;
if req.parameters.return_full_text.unwrap_or(false) {
    add_prompt = Some(req.inputs.clone());
}

let details: bool = req.parameters.details || req.parameters.decoder_input_details;

// Inference
let (response, best_of_responses) = match req.parameters.best_of {
    Some(best_of) if best_of > 1 => {
        let (response, best_of_responses) = infer.generate_best_of(req, best_of).await?
        (response, Some(best_of_responses))
    }
    _ => (infer.generate(req).await?, None),
};

// Token details
let input_length = response._input_length;


// Timings
let total_time = start_time.elapsed();
let validation_time = response.queued - start_time;
let queue_time = response.start - response.queued;
let inference_time = Instant::now() - response.start;
let time_per_token = inference_time / response.generated_text.generated_tokens;

// Tracing metadata
span.record("total_time", format!("{total_time:?}"));
span.record("validation_time", format!("{validation_time:?}"));
span.record("queue_time", format!("{queue_time:?}"));
span.record("inference_time", format!("{inference_time:?}"));
span.record("time_per_token", format!("{time_per_token:?}"));
span.record("seed", format!("{:?}", response.generated_text.seed));

// Headers
let mut headers = HeaderMap::new();


// Metrics
```

```
    // Send response

}
```

Now we take a look at the /generate route of the webserver, which appears to be the route of interest to us. The function that we care about is under the inference tag.

An important function in the Infer module is the generate_stream. The generate func just interate over the stream. It appends the request to the queue, and notify the batching backgound process.

The async fn batching_task is running in the backgound.

Notification Waiting Mechanism

```
shared.batching_task.notified().await;
```

- **Purpose**: This line waits for a signal before proceeding to batch processing. The signal (`notified()`) comes from a `Notify` instance within the shared state. It's an asynchronous wait, meaning it pauses the task without blocking the thread, allowing other tasks to run. This mechanism ensures the batching task only attempts to process requests when there's work to be done, optimizing resource utilization.

Batch Retrieval from Queue

```
while let Some((mut entries, batch, span)) = queue.next_batch(None, max_batch_prefill_tokens
```

- **Batch Composition**: The `next_batch` function attempts to assemble a batch of requests that meet specified criteria, including maximum prefill tokens (`max_batch_prefill_tokens`) and total tokens (`max_batch_total_tokens`). It returns a tuple containing request entries, the batch information, and a tracing `span` for observability.
- **Dynamic Batching**: This loop continues to retrieve and process batches as long as they are available. The dynamic nature allows for adapting the batch size based on current load and available requests.

Prefilling Batch with Context

```
let mut cached_batch = prefill(&mut client, batch, &mut entries, &generation_health).instrum
```

- **Prefill Operation**: Before sending the batch for inference, there's a `prefill` step, likely to load necessary context or initial tokens into the model. This step might involve communicating with the inference server to prepare it with the batch's context.

- **Instrumentation**: The operation is instrumented with a `span`, linking it to the broader tracing context. This helps in debugging and performance analysis by providing visibility into batch preparation steps.

Adjusting Batch Based on Feedback

- **Feedback Loop**: The subsequent loop (`while let Some(batch) = cached_batch { ... }`) handles feedback from the inference server. Based on this feedback, the function decides whether to adjust the batch size or composition. This is crucial for handling cases where the initial batch doesn't meet all stopping criteria, requiring additional tokens to be generated.
- **Batch Concatenation**: Inside this loop, if the conditions warrant (e.g., exceeding `max_waiting_tokens`), additional requests might be concatenated to the current batch to optimize throughput and resource usage. This is determined by comparing the size and token budget of the current batch against thresholds and the `waiting_served_ratio`.

Inference Call and Span Management

```
cached_batch = decode(&mut client, batches, &mut entries, &generation_health).instrument(nex
```

- **Inference Execution**: The core text generation is performed by the `decode` function, which sends the prepared batch to the inference server and awaits its response. The response may indicate that further processing is required, hence the loop.
- **Tracing and Context Propagation**: Each batch and its associated entries are linked with tracing spans (`next_batch_span`), allowing for detailed observation of the inference process. This facilitates understanding how each request progresses through the system and aids in identifying bottlenecks or issues.

Now everything is clearing up. After I read the code for Shardedclient and client, it seems that for each router, there is one sharded client, which manages a number of rust clients, each is a "client" for a GPU shard, sending and receiving grpc requests. The router calls the infer module, which batches requests from the queue, and the sharded clients copy the batch to each individual client, each sending it to the GPU Shards. The GPU Shards then coordinate using pytorch distribute framework to do the inference (I am not sure what kind of parallelism the server is using, my guess is data parallelism, but on the other hand, we see tensor parallelism in the doc).

Another thing that I am confused about is that, the router's sharded client is sending all cached batch to every client. It feels like it could have been more selective?

```
    /// Generate one token for each request in the given cached batches
    ///
```

```
/// Returns Generation for each request in batches
/// and the next cached batch
#[instrument(skip_all, fields(size = batches.iter().map(| batch | {batch.size}).sum::<
pub async fn decode(
    &mut self,
    batches: Vec<CachedBatch>,
) -> Result<(Vec<Generation>, Option<CachedBatch>, DecodeTimings)> {
    let futures: Vec<_> = self
        .clients
        .iter_mut()
        .map(|client| Box::pin(client.decode(batches.clone())))
        .collect();
    #[allow(clippy::type_complexity)]
    let results: Result<Vec<(Vec<Generation>, Option<CachedBatch>, DecodeTimings)>> =
        join_all(futures).await.into_iter().collect();
    let mut results = results?;

    let (mut generations, next_batch, mut timings) =
        results.pop().ok_or(ClientError::EmptyResults)?;

    // Merge generations from different model shards
    for (mut shard_generations, _, shard_timings) in results.into_iter() {
        generations.append(&mut shard_generations);
        // Return the timings of the slowest shard
        if shard_timings.total > timings.total {
            timings = shard_timings;
        }
    }
    Ok((generations, next_batch, timings))
}
```

Client

Nothing in particular important, the client who is sending request to the web-server (the router in the code). A good place to start to familiarize with grpc.