Microsoft

⚡ PTUs: The Go-To Solution for GenAI Apps in Production Stages

Mastering the Development of Production GenAI Applications

Pablo Salvador Lopez - AI Global Black Belt

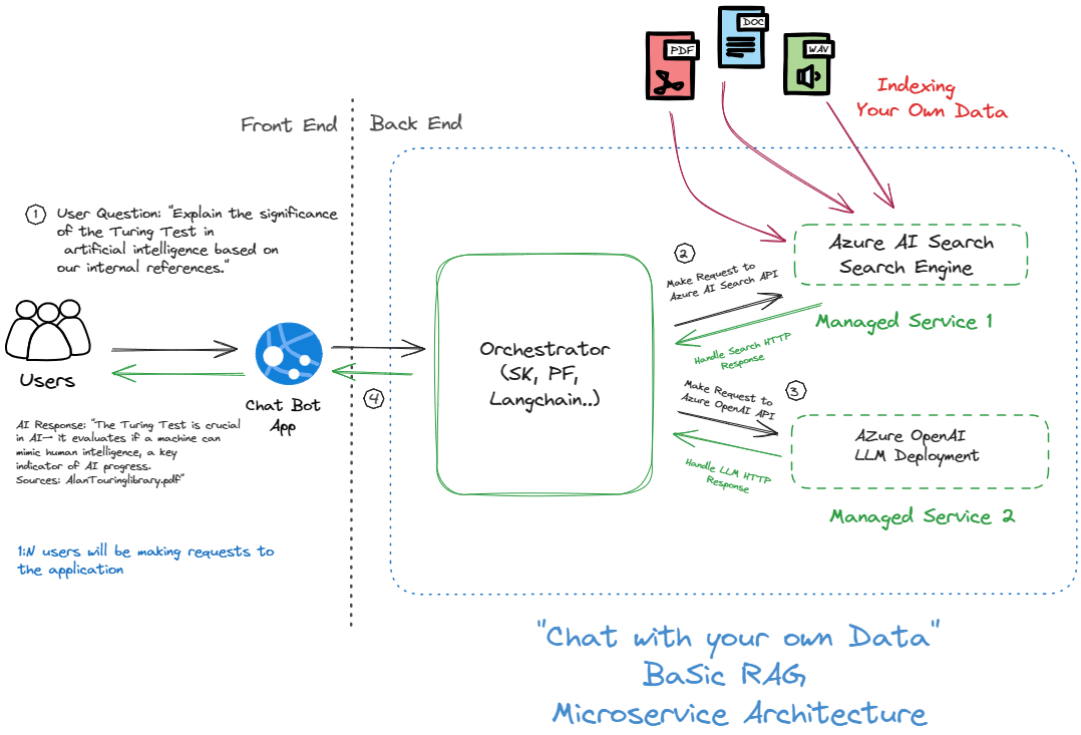
February 2024

Agenda

1. 🚀 **Demystifying the Paradigm Shift – From ML to LLM App Development**
2. 🛡️ **Challenges in Deploying GenAI Apps: Navigating Quota and Rate Limits, Understanding Latency**
    - Scalability is a system design problem.
    - Monitoring and Management is both a Platform and People problem.
    - Latency Is a Platform "Hardware" and client application design problem.
3. 🌟 **Turn Challenges into Opportunities - Stay Ahead of the Curve**
    - Adopting PTUs – Why?
    - Optimize your LLM Systems.
4. 🧮 **Calculating Your Needs: Sizing, Math, with PTUs**
    - Understanding the Calculator
    - Understanding benchmarking tool

🚀 Demystifying the Paradigm Shift – From ML to LLM App Development.

Enterprises embarking on the journey from concept to market release encounter a new landscape within the General AI Application lifecycle. The adoption of Model as a Service (MaaS) signifies a pivotal shift in their machine learning engineering operations. This approach moves away from the traditionally resource-intensive cycles of training and re-training (MLOps), managed by data science/engineering teams over the months, to a more flexible, plug-and-play, API-centric architecture.



The successful adoption of this approach is more closely aligned with the foundational principles of software engineering rather than data science. This alignment is especially pertinent when considering the design and development of microservices. Such a focus can introduce significant challenges for AI/ML/DS teams, particularly in three critical areas: **latency, scalability, and monitoring.**

Addressing these challenges effectively is essential. We have outlined the most pressing concerns faced by developers and project managers, which frequently emerge as questions in enterprise environments:

- **Scalability**: "Can my application scale to support a 1000-fold increase in user interactions with chatbots without encountering 429 'Too Many Requests' errors?"

- **Latency and Performance**: "What measures can I take to reduce latency and boost the predictive capabilities of my systems?"
- **Deployment and Monitoring**: "Which strategies are effective for centralizing deployment on a self-service platform to enhance both monitoring and management capabilities?"

These questions not only highlight specific technical challenges but also point towards the broader need for scalable, robust, and efficient AI solutions within modern software architectures.

To answer these questions well, let's explore the reasons and methods to improve your large language model (LLM) systems with AOIA as the “logic” engine.

Challenges in Deploying GenAI Apps: Navigating Quota and Rate Limits, Understanding Latency.

Scalability is a system design problem.

Scalability is intricately linked to quota and rate limits. Azure OpenAI (MaaS) manage this through two key metrics: Tokens-Per-Minute (TPM) and Requests-Per-Minute (RPM).

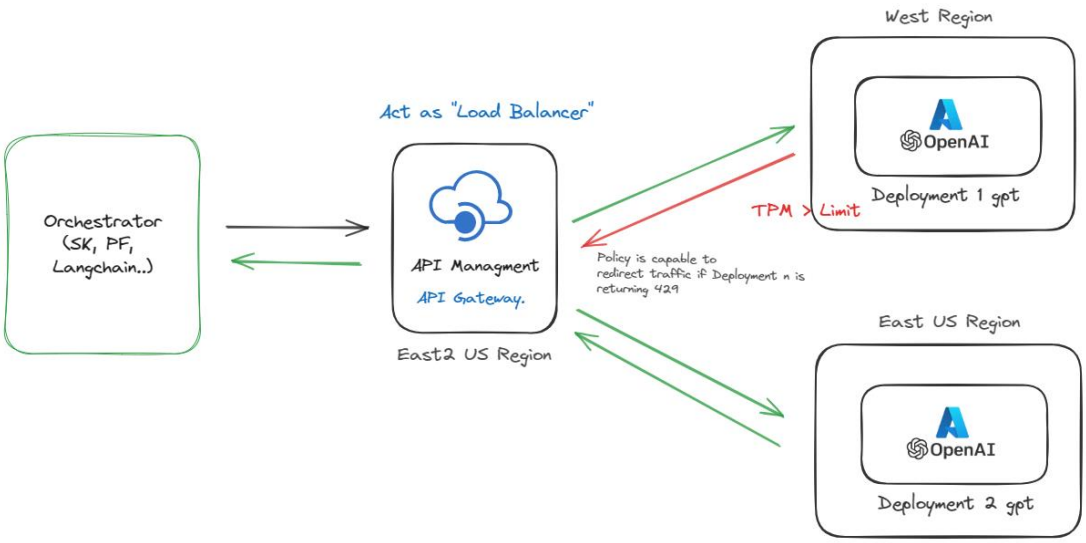
- Tokens-Per-Minute (TPM): This metric determines the volume of text your Azure OpenAI setup can process each minute. It serves as a gauge for your API's capacity, tracking text tokens and playing a vital role in how your usage is billed.
- Requests-Per-Minute (RPM): This acts as a regulatory mechanism for your API calls, functioning like a traffic light. It doesn't affect your billing directly but is essential for avoiding system overload by capping the number of requests within a given minute.

Your system's scalability is closely tied to its assigned quota limits. Employing a microservices architecture enables you to consolidate and reroute traffic across various deployments, thus enhancing your total quota capacity. Illustrated below, this diagram demonstrates how effective system design can manage scalability and quota challenges efficiently. In this setup, a gateway such as APIM, acting as a load balancer, orchestrates the distribution of traffic.

Understanding Load Balancing:

Round-Robin Load Balancing: This straightforward technique distributes client requests evenly across a pool of servers. If a server becomes unavailable, the load balancer quickly redirects traffic to the remaining operational servers. Additionally, it seamlessly integrates new servers into the rotation, ensuring a balanced load distribution. In applications like OpenAI's API, round-robin load balancing can prevent any single backend from becoming overwhelmed, thereby boosting application performance.

Smart Load Balancing : A more advanced approach, smart load balancing, dynamically assesses the current load and capacity of each server to intelligently route requests. If a server is experiencing high traffic, new requests are diverted to less burdened servers. Within an APIM context, this method can smartly allocate API calls based on backend availability and priority, maintaining responsiveness and efficiency even under fluctuating loads.



⚙ *How-to implementation of APIM with AOAI: [pablosalvador10/gbbai-azure-openai-in-production: Smart load balancing for OpenAI endpoints and Azure API Management]*

Monitoring and Management is both a Platform and People problem.

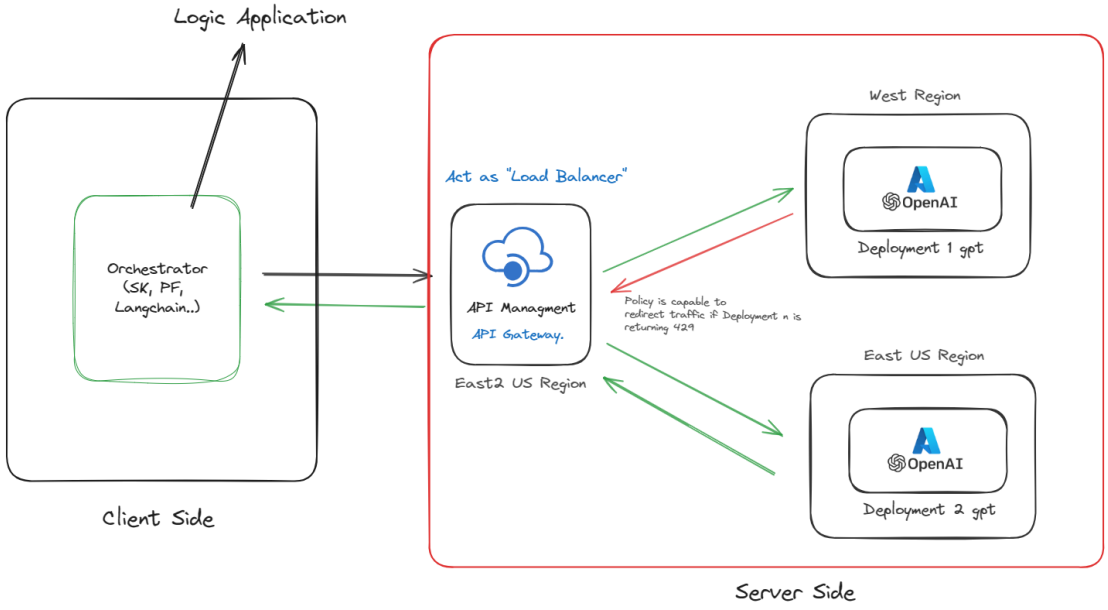
The remarkable capabilities of today's Large Language Models (LLMs), such as GPT-4, have set the stage for an exciting future with the forthcoming introduction of future iterations (let's hope GPT-5 and GPT-6 will come soon). This trajectory of relentless enhancement and sophistication in AI technology signals a shift towards exponential application growth for businesses, diverging from traditional linear advancements. This shift accentuates the critical need for immediate, strategic overhauls in management practices, deployment methodologies, and monitoring frameworks. A particular challenge has emerged from the scalability management aspect, especially when functionalities like the "dynamic quota enabled" flag are utilized without stringent oversight, allowing for asynchronous team operations to proceed without governance. An illustrative incident of this challenge was a significant budget overrun experienced by a company due to a continuous loop left unmonitored, thereby underscoring the indispensable requirement for a centralized management system tailored for enterprise-scale operations.

To navigate these complexities and to harness the potential of future exponential growth, we advocate for a centralized deployment strategy aimed at constructing an enterprise-grade General AI platform. This strategic proposition is meticulously designed to refine production

processes, augment monitoring capabilities, and adapt operational workflows. Such enhancements are pivotal for accommodating the dynamic, evolving demands of business units while also addressing cost-efficiency concerns. This comprehensive approach not only anticipates the technological advancements on the horizon but also prepares organizations to capitalize on them effectively, ensuring sustained success in an increasingly AI-driven world.

Latency Is a Platform "Hardware" and client application design problem.

Latency issues can stem from both platform hardware limitations on the server side and design aspects of client applications.



Therefore, Latency can be considered at two levels: the client level and the server level.

Understanding High Level End-to-End (E2E) Latency Process:

Latency = Network (End user to API latency) + Server (Time to process prompt tokens) + Server (Time to sample/generate tokens) + Network (API back to the end user latency)

Factors Affecting Latency and Their Possible Reasons in Client-Server Systems:

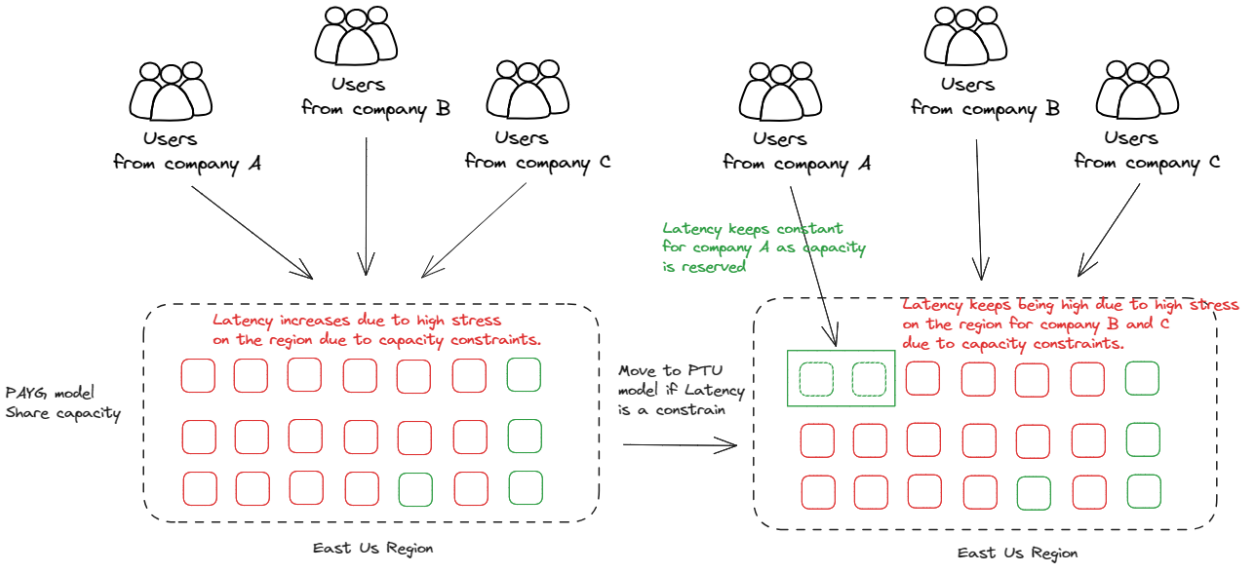
Factors Contributing to Latency	Reason	Side
Complexity of API request from user application side (especially token generation and max_tokens parameter)	Increased complexity in API requests can lead to larger data processing, thereby increasing latency	Client app
Application design issues (for example, inefficient threading causing locks)	Poor application design can lead to inefficiencies like locks which result in longer processing times	Client app
SKU type (PTU vs Pay-as-you-go)	Different SKU types have different performance characteristics. PTU offers predictable performance and reserved processing capacity	Both
Server-side processing time (based on GPU load on backend Azure Open AI servers)	Increased server-side processing time can be due to high GPU load, resulting in longer latency	AOAI service
Physical distance between client and Azure data centers	Greater physical distance can result in longer network latency	Both
Network congestion	Network congestion can result in slower data transmission, thereby increasing latency	Both

Understanding Serve-Side "Hardware" Limitations

Latency is significantly affected by hardware limitations, a reality that becomes apparent for AOAI users experiencing fluctuations in latency throughout the day in specific regions. This common challenge predominantly stems from the utilization of PAYG (Pay As You Go) deployments, which function within shared capacity environments. Each region's GPU resources are finite, and high demand in any region can dramatically affect latency, complicating the achievement of consistent, predictable performance.

AOAI SKU Options

- PAYGO (Pay-As-You-Go): This model charges based on the actual usage of the services, with no upfront commitments. Think of PAYGO as a prepaid mobile phone plan. You top up your balance and use your phone services (calls, texts, data) as needed. During peak times, you might experience slower data speeds or call quality due to network congestion (like rate limits). There's no long-term commitment, and you pay only for the services you use when you use them.
- Provisioned Throughput Units PTU offers fixed-term commitment pricing, where customers purchase a set amount of throughput capacity in order to ensure predictable and consistent service performance. PTU is like having a dedicated internet line for your home. Unlike shared connections that can slow down when many users are online, a dedicated line provides a consistent and reliable internet speed, regardless of what others in the neighborhood are doing. You pay for a specific level of capacity and performance, which is always available to you, ensuring your internet experience is always fast and uninterrupted, especially critical for high-demand applications.



Understanding Client-side Limitations.

The design of architectural decisions has a major impact on both system performance and user experience. In RAG applications, client logic that is not well designed, such as complicated reranker algorithms, can increase the overall response time, making the system less responsive. To deal with these issues, best practices and traditional optimization methods such as binary search, caching, and hash maps must be followed—they are still important in advanced LLM applications. Moreover, retry strategies and prompt engineering must be improved. Furthermore, real-time performance methods like streaming and chunking data are essential for decreasing waiting times and improving interactivity and user engagement. These methods, along with visible progress indicators, not only reduce the psychological effect of waiting but also improve system transparency and explainability. By applying these techniques strategically, developers can effectively handle system complexities, ensuring that LLM applications are both reliable and user-centric avoiding high latency.

🌟 Turn Challenges into Opportunities - Stay Ahead of the Curve

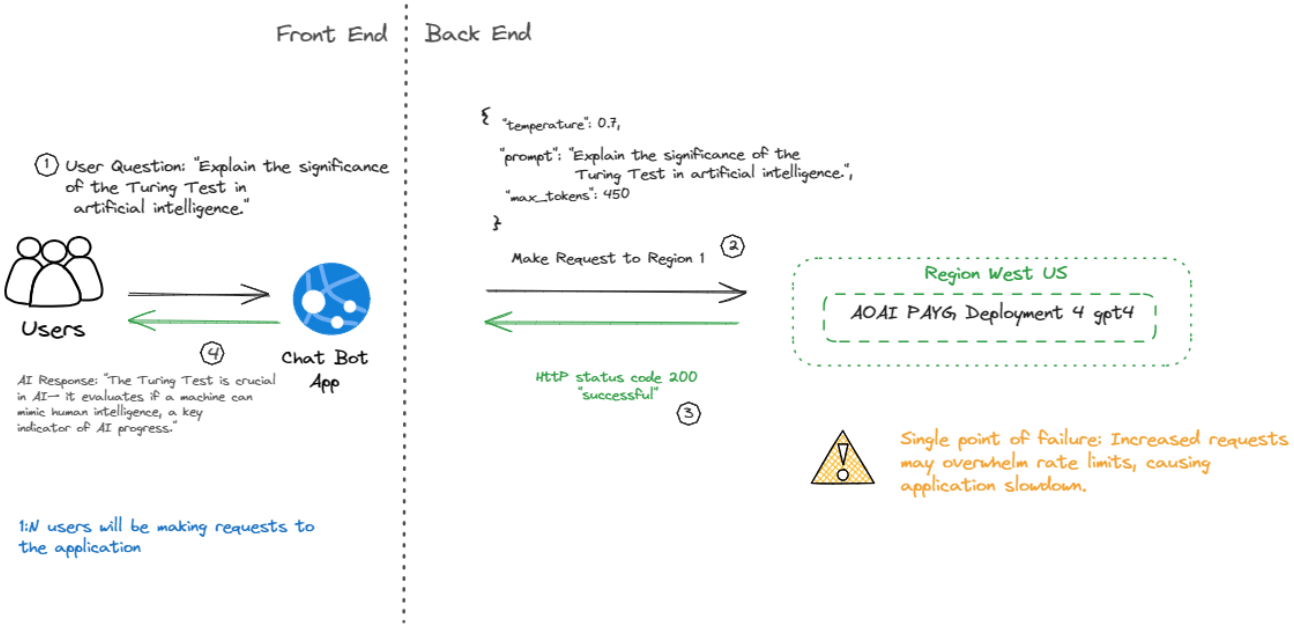
Adopting PTUs – Why?

We offer a high-level guide through each phase of your LLM application lifecycle—from Proof of Concept (POC) and Minimum Viable Product (MVP) stages to full production. This journey will help you understand the specific requirements and anticipate future challenges at each stage. By implementing a 'Smart Balance' architecture in our API Management (APIM) system, we've laid a solid foundation to meet the evolving demands of the production phase effectively (adopting PTU).

POC Stage

At the POC stage, our objective is to validate the application's value and functionality. Typically, we expose the application to a small set of users, where scalability might not be the issue. However, depending on the application's requirements, if it involves customer-facing interactions, demonstrating low latency at this stage could pose a roadblock in the PAYG model.

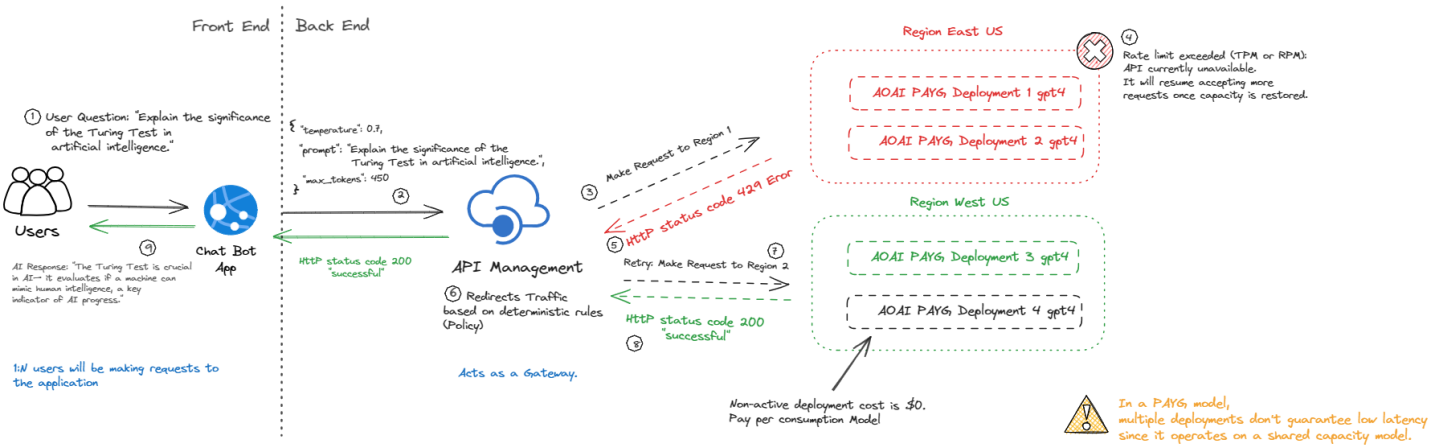
- **Deployment Type:** Single region PAYG Deployment
- **Challenges:** Scalability and Predicted Latency.



MVP Stage

In the MVP (Minimum Viable Product) phase, our goal is to enhance the scalability of our proof of concept, ensuring it can support significant growth in the user base.

- Deployment Type: Multi-region PAYG Deployments
- Challenges: While solving scalability issues, latency remains a concern as we are still reliant on shared capacity.
  Limiting the number of requests per user improves user management, yet regional usage may exacerbate latency. **We need PTUs.**



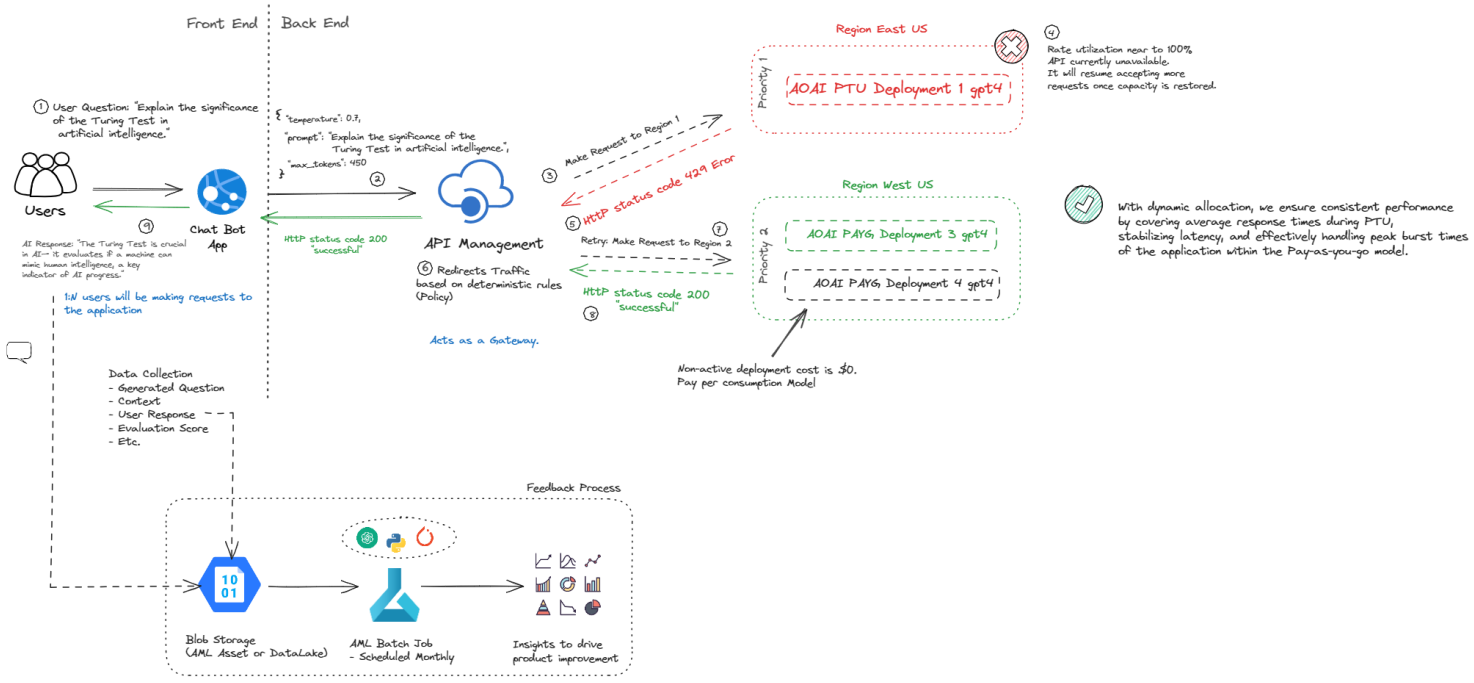
Production Stage

At the production stage, ensuring stable, predictable, and scalable performance is paramount. Our architecture is designed to meet your specific customer needs, seamlessly facilitating both external application integrations and internal operations. We prioritize flexibility and adaptability in our design through modular architecture, enabling easy customization and scalability to address future challenges and opportunities.

- **Deployment Type**: Multi-region PTU + PAYG
- **Deployments Challenges**: Finding the right balance of PTU number based on the intended
  usage to complement and optimize for the most economical model

💡 *Hint: Optimize API Rate Limit Management with PTU:*

As previously mentioned, the LLM API (AOAI) implements rate limits to manage its capacity. When these limits are exceeded, the API issues a 429 status code, accompanied by headers that detail the reason for throttling and suggest when to retry the request using the 'Retry-After' header. This header specifies the cooldown period in seconds before reattempting the request. However, PTUs provide more granular control over these parameters. By fine-tuning your application's logic to manage the 'Retry-After-ms' header more effectively, you can better utilize the backoff signal and adapt to capacity constraints. Although implementing a retry mechanism may introduce a slight increase in latency, these requests will still be processed with minimal delay, benefiting from the faster token generation rates afforded by PTU deployments.



Optimize your LLM systems!

Initially, optimization focused on minimizing latency through prompt engineering and model selection. Over time, advancements included prompt compression, fine-tuning models, and streamlining latency via caching mechanisms. Continuous improvement involved benchmarking tools, telemetry collection, and migrating to new, faster models. Client-side applications evolved from basic latency reduction to sophisticated optimization techniques, ensuring efficient processing and improved user experience.

💡 **Optimization Strategies**:

1. **Model Selection**:
    - Choosing the appropriate model for a given task is crucial for optimizing processing time. For instance, in a customer support chatbot, using GPT3.5 for simpler inquiries like FAQs can significantly reduce latency. Implementing a prompt classifier alongside can further enhance efficiency by directing complex queries to specialized models.
2. **Prompt Optimization**:
    - **Reduce Prompt Size**: Shorter prompts process faster. Techniques such as AI-based compression of user prompts could be explored,

though it is essential to validate that the meaning of the prompt remains unchanged.

- **Default Parameters**: Use default settings for **n** and **best_of** to minimize latency. (both equal to 1)
- **Output Size Reduction**: Smaller output sizes directly decrease TTLB.
- **Reducing Token Generation**: The generation of tokens usually incurs the highest latency. By halving the number of output tokens, you might approximately reduce the latency by about 50%. However, this is a heuristic and not a strict rule, as the actual impact can vary based on other factors.
- **Concise Responses**: When generating natural language, instructing the model to produce concise responses can significantly decrease the number of tokens. You might specify your request with constraints such as "under 20 words" or add an instruction like "be very brief" to guide the model's output.
- **Utilizing Limits**: Implementing parameters such as max_tokens or stop_tokens can actively control the length of the model's responses. Setting these parameters allows you to cap the output at a desired token count or terminate the response upon reaching specific terms or conditions.

3. **Streaming**:
    - Streaming requests can enhance user experience by providing real-time feedback. Consider a live translation service where text input is streamed to the model as it's being typed. This allows for immediate translation without waiting for the entire input, reducing perceived waiting time and improving responsiveness.
4. **Caching Mechanisms**:
    - Utilizing caching mechanisms can significantly improve processing time by reusing previous query results. For instance, in a news aggregation app, caching previously fetched articles can reduce load times when users revisit the same content. This enhances user experience by delivering content more quickly.
    - Use caching to save the results from previous queries that have a common prefix with the current query. KV-cache hit: The transformer takes all prompt tokens and produces one completion token at a time (one completion token per forward pass). To avoid repeating the previous keys and values for each new token produced, the transformer uses a cache (KV-cache). If two prompts, A and B, have the same prefix and come to the host one after another, the engine will use the A's prefix processing results, stored in KV-cache, instead of processing it again for prompt B. Higher KV-cache hit rate reduces input tokens processing time, does not affect output token processing time.
5. **Prompt Engineering**:

- Prompt engineering techniques focus on optimizing the structure and content of prompts for efficient processing. In an email summarization tool, condensing lengthy threads into concise summaries through prompt engineering can expedite reading and processing, saving users time and effort. https://www.promptingguide.ai/ Azure Prompt Engineering Techniques

6. **Prompt Classifiers**:
    - Prompt classifiers play a vital role in directing prompts to the most suitable processing models. For example, in an e-commerce platform, using a prompt classifier can differentiate between product inquiries and general customer service requests. This ensures that queries are directed to the appropriate models, leading to faster response times and improved user satisfaction.
7. **Prompt Compression**:
    - Prompt compression techniques aim to reduce the size of prompts without compromising accuracy. In a voice-controlled smart home device, compressing user commands can minimize data transmission and improve responsiveness, enhancing the overall user experience. GitHub - microsoft/LLMLingua: To speed up LLMs' inference and enhance LLM's perceive of key information, compress the prompt and KV-Cache, which achieves up to 20x compression with minimal performance loss.
8. **Fine-Tuning Cheaper Models**:
    - Fine-tuning cheaper models like GPT3.5 for specific tasks can significantly reduce computational costs. For instance, in a financial analysis tool, utilizing fine-tuned GPT3.5 for routine calculations and basic predictions can lead to cost savings while maintaining accuracy and performance.

Calculating Your Needs: Sizing, Math, with PTUs

Various scenarios may warrant different approaches. For instance, if your application has multiple tasks in the roadmap and operates in different time zones, PTU might be more cost-effective in the long run, especially if the pay-per-token model proves to be more expensive and challenging to manage during consumption spikes. However, the complexity lies in understanding the unique needs of your application and accurately estimating the required number of PTUs.

To determine the appropriate number of PTUs needed, a comprehensive analysis is necessary. We rely on two tools to facilitate this process:

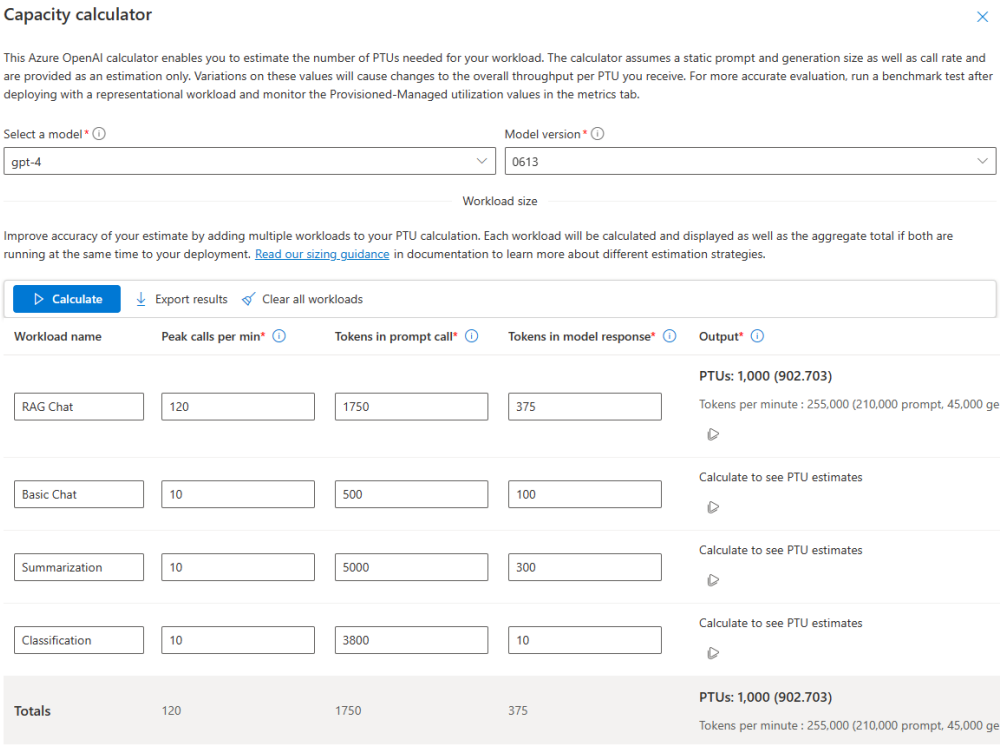
1. An internal calculator retains estimates of the PTUs needed.

2. A Python-based benchmarking tool is designed for more in-depth analyses, allowing testing of multiple scenarios within a PTU deployment.

Understanding the Calculator

The internal calculator plays a crucial role in estimating the number of PTUs needed based on various input parameters, particularly focusing on the worst-case scenario. It considers four essential metrics:

- Model Version: Identifies the specific version of the model being utilized. Different versions may have varying resource requirements and performance characteristics.
- Tokens in prompt call (Average Tokens per Inference): Represents the number of tokens in the prompt for each model call. Calls with larger prompts will consume more deployment resources. The calculator assumes a single prompt value, so for workloads with significant variance, benchmarking on actual traffic is advised to determine accurate PTU requirements.
- Tokens in model response (Tokens per Inference Generated): Denotes the number of tokens generated from each model call. Calls with larger generation sizes will utilize more deployment resources. The calculator assumes a single prompt value, so for varying workloads, benchmarking based on actual traffic is recommended to ascertain precise PTU needs.
- Peak Calls per Minute: Defines the maximum number of API calls that can be made within a minute. It reflects the peak workload the system may encounter during periods of high demand.



However, it's important to note that relying solely on these metrics may lead to an overly conservative estimate, especially when accounting for worst-case scenarios of token throughput and output tokens per minute. The calculator factors in these worst-case scenarios provide an accurate assessment of the required PTUs, ensuring optimal resource allocation at burst scenarios.

Understanding Benchmarking tool

The Azure OpenAI Benchmarking tool serves as a robust platform for evaluating the performance of provisioned-throughput deployments. Its capabilities include:

- Traffic Workload Analysis: Users can simulate different traffic patterns on their deployment to understand its performance under varying conditions. This includes assessing prompt size, generation size, and call rate to optimize deployment configurations.
- Performance Metrics: The tool generates essential performance metrics such as average and 95th percentile latencies, deployment utilization, and throughput statistics. These metrics offer insights into the responsiveness and efficiency of the deployment.
- Resource Optimization: By experimenting with different configurations, users can optimize their solution design for maximum efficiency. They can adjust parameters

like prompt size, generation size, and PTUs deployed to achieve the desired balance between performance and resource utilization.

- Deployment Validation: The tool helps validate the effectiveness of deployment configurations by providing real-world performance metrics. This enables users to make informed decisions about resource allocation and capacity planning.

Official Repository: GitHub - Azure/azure-openai-benchmark: Azure OpenAI benchmarking tool

Executing a Test: Step-by-Step Guide

Cloning Repo and Setup:

We will utilize the fork pablosalvador10/gbbai-azure-openai-benchmark: Azure OpenAI benchmarking tool (github.com), which builds on the official repository with significant contributions from both myself and Michael Tremeer. This version introduces enhancements for usability and reliability. Feel free to fork from the official repository and incorporate your custom logic as needed— but this fork provides a solid foundation for proper testing.

Python Environment Setup:

👉 gbbai-azure-openai-benchmark/SETTINGS.md at main · pablosalvador10/gbbai-azure-openai-benchmark (github.com)

How-to run:

👉 *gbbai-azure-openai-benchmark/01-ptu-benchmarking-interactive.ipynb at main · pablosalvador10/gbbai-azure-openai-benchmark (github.com)*

The system can be tested in simulated scenarios with a client pythonic, which lets you easily create and run load tests from your notebook or python scripts/command line. Test orchestration and measurement are not your concern.

- Client Initialization: The BenchmarkingTool client is created by specifying parameters like the model, region, and endpoint. This initializes the client for conducting load tests.
- Test Configuration: Parameters such as rate, duration, shape profile, number of clients, context tokens, and maximum tokens are set to define the characteristics of the load test.
- Execution: Once the client is initialized and the test parameters are configured, the run_tests() method is called to start the load test. This method triggers the execution of the test with the specified parameters.
- Load Test Process: During the test execution, the client sends requests to the specified endpoint at the defined rate. It measures the response times, success rates, and other metrics to evaluate the performance of the system under load.

- Results: After the test completes, the client may provide results such as response times, throughput, error rates, etc. These results help in assessing the performance and scalability of the system.
- Log Saving: If provided, the client may offer an option to save logs to a specified directory for later analysis or reporting.

Interpreting outcomes

Parameter	Description	Default Value	Examples
deployment	The name of the Azure OpenAI deployment.	N/A	"my-openai-deployment"
api_base_endpoint	The base endpoint URL of the Azure OpenAI deployment.	N/A	"https://my-openai.azure.com"
api_version	The version of the OpenAI API to use.	"2023-05-15"	"2023-05-15"
api_key_env	The environment variable that contains the API key.	"OPENAI_API_KEY"	"MY_OPENAI_API_KEY"
clients	The number of parallel clients to use for generating load.	20	50

requests	The total number of requests to make during the load run.	'until killed'	1000
duration	The duration in seconds for which to run the load test.	'until killed'	3600 (1 hour)
run_end_condition_mode	Determines the end condition for the run based on requests and duration.	'or'	'and'
rate	The rate of request generation, in Requests Per Minute (RPM).	As fast as possible	200
aggregation_window	The duration in seconds of the statistics aggregation window.	60	120
context_generation_method	The method to generate context messages for testing.	"generate"	"replay"
replay_path	The path to a JSON file containing	N/A	"/path/to/replay_file.json"

	messages for replay.		
shape_profile	Defines the shape of requests.	"balanced"	"custom"
context_tokens	The number of context tokens to use when shape_profile is "custom".	N/A	256
max_tokens	The maximum number of tokens requested when shape_profile is "custom".	Unset	1024
prevent_server_caching	Whether to add random prefixes to all requests to prevent server-side caching.	True	False
completions	The number of completions to request for each API call.	1	5
frequency_penalty	Adjusts the likelihood of the model repeating the same	N/A	0.5

	line verbatim.		
presence_penalty	Adjusts the likelihood of the model talking about new concepts.	N/A	0.5
temperature	Controls randomness in generation, with higher values leading to more random outputs.	N/A	0.9
top_p	Controls the diversity of generated content by only considering the top p% of probabilities.	N/A	0.9
output_format	The format of the output from API calls.	"human"	"json"
log_save_dir	Directory to save logs if provided.	N/A	"/logs/my_run_logs"
retry	The strategy for retrying	"none"	"exponential_backoff"

	failed requests.		

Understanding Results: Interpretation Guidelines

```
2024-03-11 07:28:35 INFO     rpm: 75.0  processing:
5    completed: 95    failures: 0    throttled: 0    requests:
95    tpm: 93900.0 ttft_avg: 0.675  ttft_95th: 0.927  tbt_avg:
0.072  tbt_95th: 0.079  e2e_avg: 18.933 e2e_95th: 20.569
context_tpr_avg 999  gen_tpr_10th 253  gen_tpr_avg
253  gen_tpr_90th 253  util_avg: 0.3%   util_95th: 0.6%
2024-03-11 07:28:38 INFO     rpm: 75.9  processing:
0    completed: 100   failures: 0    throttled: 0    requests:
100   tpm: 95088.0 ttft_avg: 0.675  ttft_95th: 0.924  tbt_avg:
0.072  tbt_95th: 0.079  e2e_avg: 18.908 e2e_95th: 20.522
context_tpr_avg 999  gen_tpr_10th 253  gen_tpr_avg
253  gen_tpr_90th 253  util_avg: 0.3%   util_95th: 0.7%
2024-03-11 07:28:38 INFO     rpm: 75.9  processing:
0    completed: 100   failures: 0    throttled: 0    requests:
100   tpm: 95088.0 ttft_avg: 0.675  ttft_95th: 0.924  tbt_avg:
0.072  tbt_95th: 0.079  e2e_avg: 18.908 e2e_95th: 20.522
context_tpr_avg 999  gen_tpr_10th 253  gen_tpr_avg
253  gen_tpr_90th 253  util_avg: 0.3%   util_95th: 0.7%
2024-03-11 07:28:38 INFO     finished load test
```

- Timestamps: The data spans from 07:27:42 to 07:28:25 on March 11, 2024.
- RPM (Requests Per Minute): This metric shows the number of requests processed per minute. It varies throughout the run, starting around 20 RPM and increasing steadily to around 68 RPM. Successful Requests Per Minute. Note that it may be less than --rate as it counts completed requests.
- Processing: Indicates the number of requests being processed at any given time. It fluctuates slightly but generally increases as the run progresses.
- Completed: Represents the number of requests successfully completed. It increases over time, reflecting the successful processing of requests.
- Failures: Indicates the number of failed requests. In this run, there are no reported failures, which is desirable.
- Throttled: Shows whether any requests were throttled. Throttling occurs when the system limits the number of requests it processes within a given time frame. In this run, there are no instances of throttling.

- Requests: The total number of requests made. It increases steadily throughout the run. Deprecated in favor of completed field (output values of both fields are the same).
- TPM (Tokens Per Minute): Similar to RPM but may refer to higher-level Tokens rather than individual requests. It shows how many tokens are being processed per minute.

Latency Metrics:

- **ttft_avg** (Time To First Token Average): The average time taken from sending a request to receiving the first Token of the response.
- **ttft_95th** (Time To First Token 95th Percentile): The value below which 95% of the time to first Token measurements fall. This is an indicator of response time for most requests.
- **tbt_avg** (Time Between Tiers Average): Average time spent between different tiers or components of the system.
- **tbt_95th** (Time Between Tiers 95th Percentile): The value below which 95% of time between tiers measurements fall.
- **e2e_avg** (End-to-End Average): Average end-to-end response time.
- **e2e_95th** (End-to-End 95th Percentile): The value below which 95% of end-to-end response time measurements fall.

Throughput Metrics:

- **context_tpr_avg** (Context Throughput Rate Average): Average throughput rate at the context level.
- **gen_tpr_10th, gen_tpr_avg, gen_tpr_90th** (Generic Throughput Rate): Percentile values for throughput rate.
- **util_avg (Utilization Average):** Average system utilization.
- **util_95th (Utilization 95th Percentile):** 95th percentile of deployment utilization percentage as reported by the service.