Seoul, Korea
siksicking@gmail.com

Sunsik Kim

Data Scientist

Portfolio: sunsik.dev/en
linkedin.com/in/sunsik-kim

Data Scientist with 5 years at Bunjang (Korea's leading C2C marketplace, 10M+ MAU), designing and building the model side of production ML/LLM systems that solve real user problems, from in-chat abuse detection to product recommendation. Looking to apply that problem-solving to services with a much larger user base and make a bigger impact.

Work Experiences

Data Scientist Bungaejangter Inc. (Bunjang) 2021 — Present

Projects

In-Chat Abuse Detection System **December 2025 — July 2026**
Bunjang *Seoul, Korea*

- **Result**: Delivered a **4.6x increase in abuse detection coverage** over the legacy regex/OCR system, and **49.5% more sanctioned users** after full rollout with false-positive rate held within threshold — driving sanctioned users' subsequent transactions to escrow and capturing GMV previously occurring off-platform.
- **System**: Implemented the batch producer and LLM worker in an **event-driven inference architecture** on AWS, where the batch producer queries chat logs via **Athena** and publishes detected events to **SQS** for consumption by LLM workers running on an **EKS** cluster, built for a future batch-to-real-time migration.
- **Model**: Designed a **4-stage cascaded LLM inference pipeline**, decomposing a monolithic instruction into task-specific stages to lift precision on production-representative test sets; selected **gemma-4-26B-A4B-it** after benchmarking 5 open-source candidates, replacing the external LLM API with a **managed-hosting endpoint** to bring operating costs within budget.
- **Infrastructure**: Benchmarked a **self-hosted vLLM deployment** on an AWS **DLAMI g7e.2xlarge** GPU instance against the incumbent managed LLM hosting service — same pipeline, both setups — showing a **60% wall-time reduction** at precision parity within margin of error, the quantitative basis for migration planning.

AI-Assisted Product Registration System **January 2025 — June 2025**
Bunjang *Seoul, Korea*

- **Result**: Versus a non-AI control cohort, drove a **4pp increase in weekly registration conversion** and cut time-to-first-listing of new users by 13.5%. Among AI-assisted users, **listings per user rose 47.2%** post-launch, and 85% of survey respondents called the feature helpful.
- **System**: Shipped a product registration system with **field-specific components** (title, category, description, price, etc.), each orchestrating its own model and downstream service by input/output modality (image, text, multimodal).
- **Model**: Used an LLM for open-ended language tasks (title, description), and fine-tuned a lightweight in-house classifier for category from an existing Korean text encoder, tuned to match the four-slot UI carousel (HitRatio@4) — **reserving the LLM for tasks that actually needed it**.
- **Pipeline**: For price — the one field needing signals from comparable listings, not just item attributes — built a **multi-stage pipeline** chaining an LLM (query generation), an in-house search engine, and a reranking-and-statistics service to derive a range from live comparables.

Similar Product Recommendation System **March 2024 — October 2025**
Bunjang *Seoul, Korea*

- **Result**: Drove **10.2% CTR / 13% PDP-entry gains** on the home recommendation surface, and **4.1% CTR / 9% PDP-entry gains** on the product-detail surface after full rollout; PDP-entry user share rose **10.4%** and **3.6%** respectively.
- **System**: Designed and implemented a **hybrid product encoding scheme** unifying title (text encoder), category-brand (collaborative filtering on user-item interactions), and price/size (sinusoidal positional encoding) into a single embedding for real-time similarity search over a multi-million-item catalog.
- **Model**: **Pretrained a lightweight Korean text encoder** from scratch on Bunjang product text with **ELECTRA** via PyTorch/Hugging Face transformers, using **multi-GPU DDP** training (4×A100, Hugging Face Accelerate) on an on-prem workstation, then **supervised SimCSE** domain alignment on (search query, product title) pairs — achieving retrieval parity with a 20x larger public-model baseline at **4–5x lower CPU cost per request** on AWS Graviton (c7g.large).
- **Infrastructure**: Implemented the **hourly indexing batch job** that encodes new and updated listings into product vectors for Aurora **pgvector**, triggered as a **SageMaker** Training Job via MWAA-orchestrated Airflow. Built the **embedding API** as a containerized **FastAPI** service, pushed to ECR and deployed on EKS.

Education

Master of Statistics, *Korea University* 2019 — 2021
Bachelor of Economics (Statistics), *Sungkyunkwan University* 2013 — 2019

Project Details

In-Chat Abuse Detection System **December 2025 — July 2026**

Problem

Bunjang runs an escrow payment system that charges a selling fee, and sellers were evading it by sharing contact info or bank details in chat to settle off-platform. The existing regex-based system, matched against known abuse patterns, couldn't keep up as sellers kept obfuscating further (e.g., "0iI0-46pal739Oil"). We needed an LLM-based system that could interpret chat context to catch evasion, meeting the requirements below — which I did by **building a 4-stage LLM inference pipeline**, described next.

- **Multimodality**: Abuse information is often conveyed inside images as well as text, so the system had to be capable of both text and image reasoning.
- **Precision over recall**: Wrongly sanctioning a legitimate user immediately damages customer experience, which is judged to be riskier than missing a detection. So I prioritized precision over recall during evaluation.

Action 1. Building the LLM Inference Pipeline

To implement the LLM inference pipeline, I worked through the following steps.

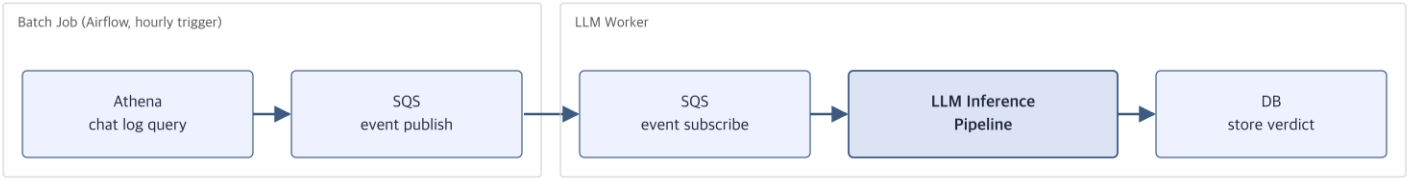
1. **Building test data**: Working with the customer support team that monitors and sanctions abusers, I clarified internal sanction rules to be applied and jointly built test data to evaluate the LLM system.
2. **Prompt engineering**: I encoded the rules as a prompt and iterated using precision/recall on the test data, but exceptions and fine-grained rules interfered with each other, blocking production-level precision.
3. **Building a task pipeline**: I replaced the single-prompt architecture with one that splits sanction judgment into 4 independent subtasks, each with its own prompt and evaluation criteria, resolving the interference.

This resulted in a 4-stage **LLM inference pipeline** — exception-rule judgment, image review, abuse-signal detection, and identifier-sharing detection — that brought precision to the target level.

Action 2. Building the LLM Inference Environment

Applying this pipeline to every chat exceeded our budget on a token-billed API, so I explored an alternative to host an open-source LLM via a managed service instead, as follows.

1. **Selecting an open-source LLM**: Budget required a sub-30B model to be run on a single instance, so I benchmarked precision and throughput across five sub-30B models released as of Q2 2026 and adopted gemma-4-26b-a4b-it, hosted on that service's instance.
2. **Batch job and pipeline implementation**: Within the inference flow diagrammed below, I owned building the Batch Job that collects chat data to be reviewed, and the inference pipeline inside the LLM Worker.
3. **Benchmarking a self-hosting migration**: After the system was rolled out to production, the hosting service announced hardware end-of-life and a forced upgrade by year-end, risking lock-in to whatever pricing followed. To keep a runtime our team could control, I benchmarked self-hosting the same model via vLLM on an AWS GPU instance. Self-hosting also let me choose quantization freely, cutting per-task latency by up to 60% on average with no quality loss.



Result

1. **Increased abuse-detection volume**: To prove the system caught abuse the legacy OCR/regex system missed, I ran both in parallel and compared logged detection events, confirming **4.6x more** abuse attempts detected.
2. **Low false-positive rate**: Defining false positives as approved appeals from sanctioned users, I confirmed the daily rate stayed consistently **under 5%**, letting us detect more abuse safely and fully replace the legacy system.
3. **Increased escrow transaction value**: After full rollout, users sanctioned without a successful appeal rose **49.5%**, and some subsequently switched to escrow payment, **meaningfully growing transaction value** by bringing previously off-platform transactions back onto it.

AI-Assisted Product Registration System **January 2025 — June 2025**

Problem

Many users entered the product listing screen but dropped off before completing it, keeping registration conversion lower than expected. The manual burden of filling required fields — title, category, description — was suspected as the main cause. I **introduced an AI-assisted feature that auto-generates and recommends values for these fields** to improve conversion. Since each field has a different input/output modality, I designed a separate model and calling scheme per field.

Action 1. Building the Text-Generation Components (Title, Description)



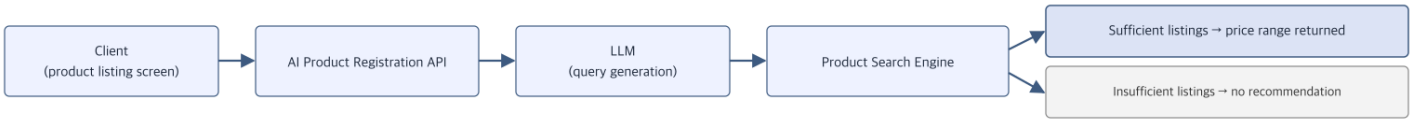
1. **Title Generation** (image → text): To minimize latency between image upload and title generation, I used only the first thumbnail image. The generated title surfaces as an AI-suggestion badge above the keyboard rather than auto-filling the field — tapping adopts it, ignoring it lets the user keep typing — so the existing registration experience stays intact.
2. **Description Generation** (multimodal → text): Generated from image, title, and category combined. To keep it from overwhelming the listing screen, I specified a 4-sentence convention in the prompt — a title-like opener, two sentences on visible features, a closing purchase-encouraging line — to control length.

After launch, I created a **dashboard to monitor ROUGE-L scores** between generated output and the text actually registered by category, giving a rough sense of adoption and how much of the original survived when adopted.

Action 2. Building the Structured-Field Recommendation Component (Category)

- **Using a text classifier**: Category classification has a narrow domain and formulaic phrasing, so instead of using an LLM I fine-tuned a lightweight text encoder — pretrained on Bunjang's text data — on (title, category ID) pairs. The resulting API returns 4 category IDs from a title and is called directly by the client, saving LLM tokens.
- **Retraining against data drift**: I retrained the classifier on a fixed schedule so recommendations wouldn't go stale for titles using new terminology, plus built a manual trigger for retraining on category additions/removals.

Action 3. Building the Price-Recommendation Component



1. **Query generation**: Since the price had to reflect the item's current market price, I had an LLM generate "a search query likely to surface products with this category and title" and fed it into Bunjang's search engine.
2. **Representative-price estimation**: The representative price comes from the price distribution of same-category search results, but too few results make the estimate's variance untrustworthy. So I recommend a price only when results are sufficient.
3. **Returning a price range**: When sufficient, I rerank results by title similarity and sold status, then return a price range based on the top-ranked products' median price.

Result

I've prepared a demo video showing how this feature assists registration. Comparing 2 weeks before and after launch, I observed:

1. **Increased registration conversion**: Weekly average conversion rose **4pp** for the AI-assisted group versus non-AI.
2. **Increased listings per user**: Listings per AI-assisted user rose **47.2%** post-launch.
3. **Reduced time to first listing**: Time to first listing for new AI-assisted signups **shortened by 13.5%**.

These changes show the feature substantively increased listing activity along two axes: **accelerating first listings for new users** and **activating repeat listings for existing ones**.

Similar Product Recommendation System **March 2024 — October 2025**

Problem

Engagement on the product-detail page's recommendation surface was lower than expected, and the collaborative-filtering algorithm running there — recommending "products other users viewed together" — was identified as the cause: it didn't fit Bunjang users' pattern of already knowing what they want and comparing similar alternatives before buying. We instead needed a content-based algorithm recommending products similar to the one clicked. To meet the requirements below, I **built a content-based recommendation system around a product vector encoder that could run on CPU instances alone**.

- **Vector-based recommendation**: To quantify similarity between products, I needed encoding logic that converts diverse attributes — title text, category, brand, price — into vectors where similarity can be computed for each.
- **Operating-cost optimization**: Recommendation inference runs 24/7 with high call volume, so a system that could run stably on minimal resources was preferred to keep costs down.

Action 1. Building the Product-Attribute Encoders (Title, Category, Brand, Price, etc.)

Each attribute has a different nature, so I devised a different vectorization approach for each.

1. **Title Encoder**: Public pretrained text encoders don't reflect Bunjang's domain vocabulary, so I needed to train on domain data regardless. I referenced **RoBERTa** to implement a pretraining task, training directly on Bunjang's product text.
2. **Category-brand Embedding**: I replaced each item in user-item interaction data with its (category, brand) pair, then trained a **collaborative-filtering algorithm** mapping user and pair embeddings into a shared space, keeping only the pair embeddings as features.
3. **Price Encoding**: Price can be compared numerically, but I preferred a method where similarity is measured in a single vector operation. So I used the **positional encoding formula** from the Attention paper (Vaswani, 2017) to convert values into vectors whose similarity changes monotonically with the difference.

I normalized each per-attribute vector, weighted it by domain intuition, and concatenated them into a single product vector.

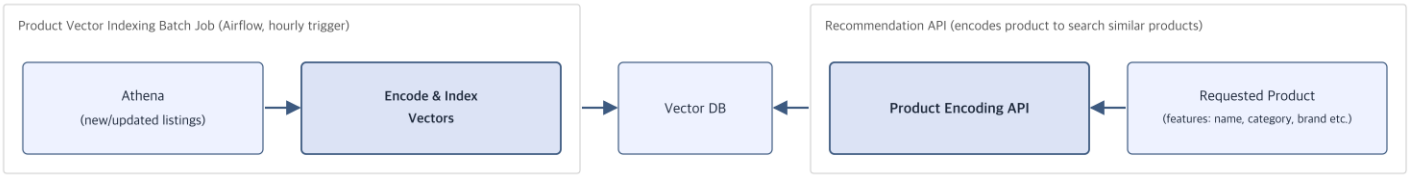
Action 2. Improving the Title Encoder

The initial encoder's tokenizer depended on torchtext. After its discontinuation, the training code needed a full rewrite to replace the tokenizer, which I used as a chance to improve it in two ways.

1. **Swapping the pretraining task**: RoBERTa's MLM task uses only 15% of tokens; **ELECTRA**'s RTD task uses every token as a signal, better suited to our modest tens-of-gigabytes dataset. I implemented the pretraining logic myself, training with **PyTorch DDP** on 4×A100 GPUs using the Hugging Face Accelerate library.
2. **Adding an extra training task for domain alignment**: After pretraining, I collected (search query, title) pairs from a year of search-then-click logs and ran **supervised SimCSE training**, so that semantically similar text maps to nearby embeddings.

Action 3. Building the Product Vector Indexing Batch Job and Encoding API

I implemented the Batch Job and the embedding API that make up the recommendation system diagrammed below.



Result

A/B testing this algorithm against the existing collaborative-filtering one showed a **30% higher average daily CTR** on the product detail page, replacing it. It now also powers a home-feed slot recommending similar products; see the demo video of the feed updating in real time on a click. The subsequent title-encoder improvement produced another round of gains across both surfaces:

1. **Home feed**: total clicks in the recommendation area rose 13% versus before, a **10.2% CTR increase**
2. **Product-detail page**: total clicks in the recommendation area rose 9% versus before, a **4.1% CTR increase**
3. **Increased users engaging with recommendations**: the share of users clicking into the recommendation area rose **10.4% and 3.6%** for the home feed and product-detail page respectively