

KURO::VISION

Sovereign Reasoning-Guided Image Generation Architecture

GPU **RTX 5000 Ada** VRAM **32GB** RAM **64GB** CPU **12-core** Provider **TensorDock**

CONTENTS

- 01 Gemini / Nano Banana Pro — Architecture Analysis
- 02 KURO::VISION — Sovereign Equivalent Design
- 03 VRAM Budget & Model Selection
- 04 Pipeline Architecture
- 05 Red Team Analysis
- 06 Post-RT Mitigations & Refined Design
- 07 Implementation Plan

SECTION 01

Gemini / Nano Banana Pro — Architecture Analysis

Nano Banana Pro is the marketing name for **Gemini 3 Pro Image** — not a standalone image model, but a unified multimodal transformer that natively reasons across text and pixel space. Understanding its architecture is critical because it reveals both what makes it powerful and where the seams are that we can exploit with open-source components.

Core Architecture: Unified Token Space

Gemini 3 Pro uses a **single decoder-style transformer backbone** with learned encoders for each modality. Text, images, audio, and video are converted into a shared token space, then reasoned over jointly. This is fundamentally different from "gluing a vision model to a language model."

Key components:

Vision Encoder — ViT-style backbone converts pixels into patch embeddings, projected into the same embedding space as text tokens. The model doesn't "see" images separately — it reasons over them as first-class tokens alongside language.

MoE Routing — Mixture-of-Experts layers scale capacity sparsely, activating only relevant experts per token. This is how Google gets a massive model to run at acceptable latency.

Specialized Output Heads — Render responses as text, JSON, tool calls, code, or image tokens. The image output head drives a diffusion-like renderer (GemPix 2) that converts semantic embeddings into pixels.

The "Thinking Mode" — Why It Matters

This is Nano Banana Pro's killer feature and the key architectural insight we need to replicate. Before generating a single pixel:

1

Prompt Analysis

The Gemini backbone parses the prompt for semantic logic, physical causality, spatial relationships, and emotional intent. It doesn't just match keywords — it builds a structured understanding.

2

Interim "Thought Images"

The model generates up to 2 draft images internally to test composition and logic. These are visible in the backend trace but not billed. The final thought image becomes the rendered output.

3

Search Grounding (Optional)

If the prompt references real-world data, the model can call Google Search as a tool, retrieve facts, then generate imagery grounded in live data. This is a tool-use capability, not magic.

4

Final Render

The GemPix 2 rendering engine synthesizes the final image from the reasoning-enriched token sequence. Resolution up to 4K. SynthID watermark applied.

What Makes It Hard to Replicate

CAPABILITY	WHY IT'S HARD	OPEN-SOURCE GAP
Unified token space	Text and images share the same embedding space — reasoning crosses modality boundaries seamlessly	No equivalent
Thought images	Draft-and-refine loop happens within a single forward pass, not a separate pipeline	Emulatable
Text rendering	The LLM backbone understands language, so rendered text is spelled correctly by construction	Partial (LoRA)
Search grounding	Just tool use — an LLM calling a search API before generating	Fully replicable
Conversational editing	Multi-turn context maintained via thought signatures	Replicable
4K native output	Custom rendering head trained at scale	Via upscaler chain
Character consistency	Identity embeddings maintained in token space	Via LoRA / IP-Adapter

The Honest Truth

CRITICAL INSIGHT

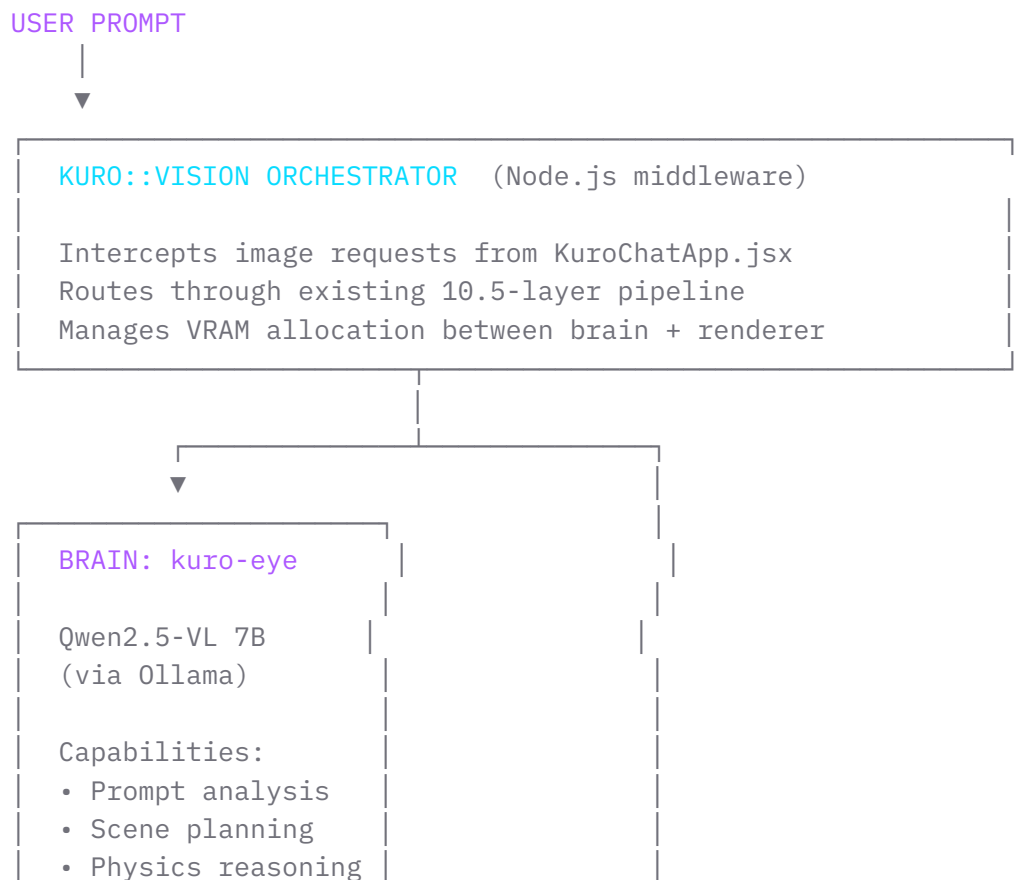
You cannot build a true Nano Banana Pro equivalent with current open-source tools. Google trained a trillion-parameter MoE model on exabytes of multimodal data. That's not something

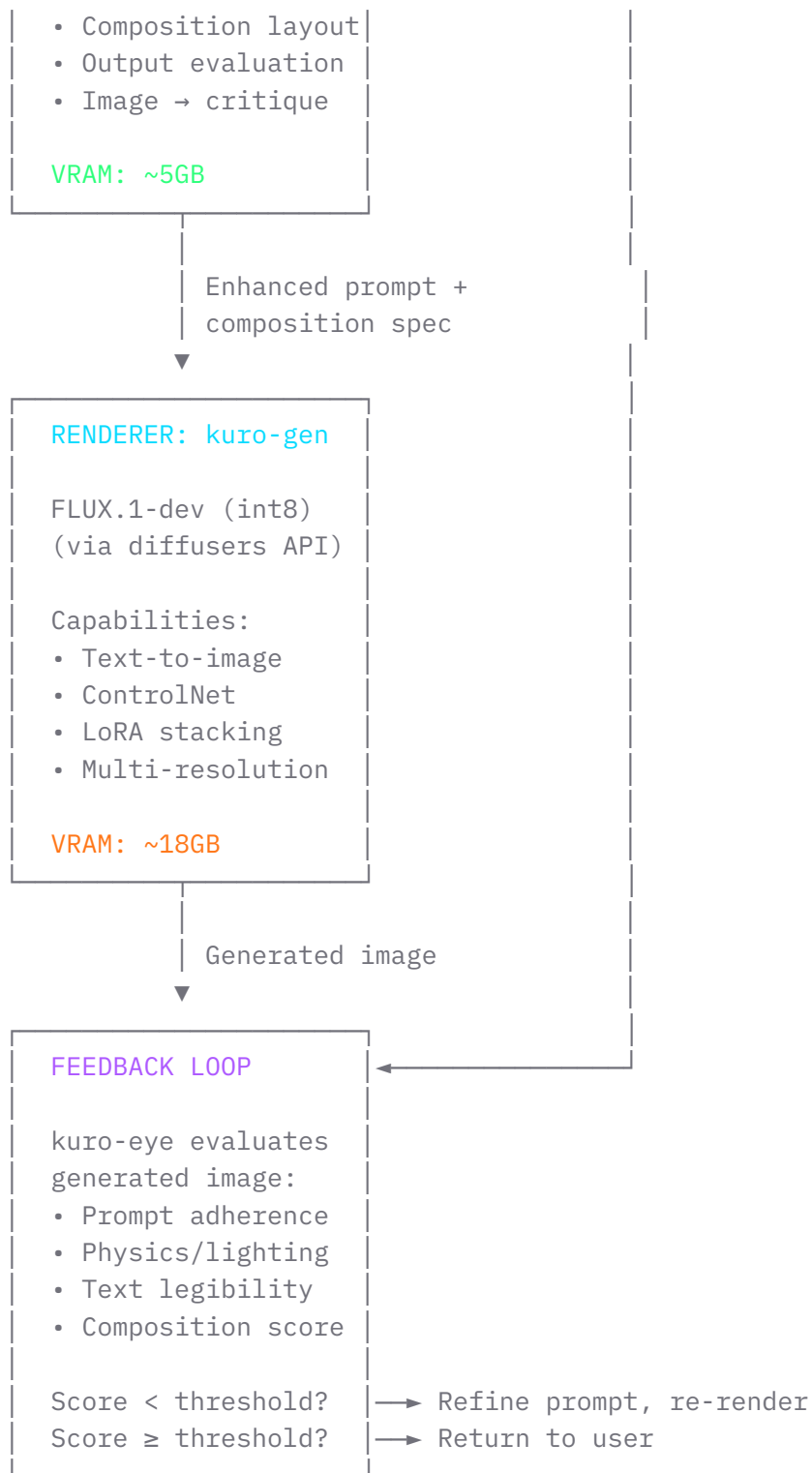
you replicate on a 32GB GPU. **But you don't need to.** Nano Banana Pro is a general-purpose model that's good at everything. A **sovereign pipeline of specialized components** — where a reasoning brain directs a best-in-class diffusion model — can match or exceed it in specific domains while maintaining the same user-facing experience: type a natural language prompt, get a reasoned, high-quality image.

SECTION 02

KURO::VISION — Sovereign Equivalent Design

The core insight: decompose Nano Banana Pro's unified architecture into a **two-brain pipeline** where a vision-language model acts as the "art director" and a diffusion model acts as the "renderer." Connected by a KURO orchestration layer that handles prompt engineering, feedback loops, and VRAM management.





Why This Works

ARCHITECTURAL ADVANTAGE

Nano Banana Pro's "thinking mode" generates 1-2 draft images internally. Our pipeline does

the same thing explicitly — kuro-eye reasons about the prompt, kuro-gen renders, kuro-eye evaluates, and if quality is insufficient, the loop refines and re-renders. The user sees the same thing: a brief "thinking" phase followed by a high-quality image. The difference is latency (Google's is faster because it's unified), but the output quality can match or exceed in specialized domains.

Key parity features: Reasoning-guided generation ✓, draft-and-refine ✓, search grounding (via kuro-main web tools) ✓, conversational editing (via session memory) ✓, multi-turn context ✓

SECTION 03

VRAM Budget & Model Selection

RTX 5000 Ada gives us 32GB to work with. Here's how it allocates across operating modes:

Mode A: Simultaneous (Both models loaded)

kuro-eye — Qwen2.5-VL 7B Q4_K_M	~5GB
5GB	
kuro-gen — FLUX.1-dev int8	~18GB
18GB	
KV Cache + Overhead	~4GB
4GB	
Free headroom	~5GB
5GB free	

VERDICT: FITS

Total: ~27GB / 32GB. Both models coexist in VRAM simultaneously. No model swapping needed. This is the optimal configuration — zero latency penalty from loading/unloading. The

5GB headroom handles ControlNet adapters, LoRA weights, and VAE operations.

Mode B: Heavy Reasoning (Upgrade path)

kuro-eye — Qwen2.5-VL 32B Q4	~20GB
20GB	
kuro-gen — FLUX.1-dev int4	~13GB
13GB — swapped in	

VERDICT: SWAP REQUIRED

Total: ~33GB — exceeds 32GB. Requires model offloading: kuro-eye reasons first (20GB), then offloads to system RAM (64GB available), kuro-gen loads for rendering (13GB). Adds ~8-12s swap latency. Worth it only if 7B reasoning is insufficient for your prompts.

Model Selection Rationale

ROLE	MODEL	WHY THIS ONE
kuro-eye Brain	Qwen2.5-VL 7B	Best open-source VLM at this size. Outperforms GPT-4o-mini on vision tasks. Can see AND reason about images — critical for the feedback loop. Runs via Ollama. Fits comfortably alongside FLUX.
kuro-gen Renderex	FLUX.1-dev 12B	SOTA open-source diffusion model. Superior prompt adherence to SD3/SDXL. Strong text rendering (better than any diffusion model pre-2025). 12B params with int8 quant fits in 18GB. Massive LoRA ecosystem.
kuro-main Chat (existing)	DeepSeek-R1 8B / Devstral	Your existing KURO chat brain stays separate. kuro-eye is a new addition specifically for vision tasks. kuro-main handles general chat; when an image request is detected, the orchestrator routes to kuro-eye + kuro-gen.

Pipeline Architecture

How a user prompt flows through KURO::VISION, mapped to your existing 10.5-layer pipeline:

- 0 Semantic Router (L3) detects image intent**
User says "generate an image of..." or "create a picture..." → L3 flags intent as IMAGE_GEN.
Routes to KURO::VISION orchestrator instead of standard chat flow.
- 1 kuro-eye: Prompt Analysis & Scene Planning**
The VLM receives the user prompt + system instructions to decompose it into: subject, composition, lighting, camera angle, mood, color palette, text requirements, physics constraints. Outputs a structured JSON scene description + an optimized diffusion prompt.
- 2 kuro-eye: Prompt Enhancement**
The raw user prompt "make me a poster for my coffee shop" becomes: "A professional coffee shop poster, warm ambient lighting, shallow depth of field, rich brown and cream color palette, modern sans-serif typography reading 'FRESH BREW DAILY', artisan latte art in ceramic cup, clean minimalist layout, 3:4 aspect ratio, commercial photography style"
- 3 kuro-gen: Image Synthesis**
FLUX.1-dev receives the enhanced prompt + any LoRA weights (style, subject) + ControlNet conditions (if provided). Generates at 1024×1024 base resolution. ~15-30 inference steps.
- 4 kuro-eye: Quality Evaluation (Feedback Loop)**
The VLM evaluates the generated image against the original prompt. Scores: prompt adherence (0-10), text legibility (0-10), composition quality (0-10), physics accuracy (0-10). If average < 7, generates a refinement prompt and triggers re-render (max 2 refinements).
- 5 Post-processing & Delivery**
Optional: Real-ESRGAN upscale to 2K/4K. Image saved to /var/www/kuro/data/images/. Served to KuroChatApp.jsx as inline image in chat stream. Session memory stores the generation context for conversational editing.

Conversational Editing (Multi-turn)

After initial generation, the user can say "make the background darker" or "change the text to say MORNING BREW." The orchestrator:

1. Pulls the previous generation context from session memory
2. kuro-eye interprets the edit instruction relative to the existing image
3. Generates a modified prompt preserving unchanged elements
4. kuro-gen re-renders with the same seed + modified prompt (or uses img2img with the previous output as init image for precise edits)

SECTION 05

Red Team Analysis

Systematically attacking every assumption in this design. If it survives Red Team, it ships.

RT-01: LATENCY KILL

Attack: Two-model pipeline is inherently slower than Nano Banana Pro's single-pass generation. Gemini generates in ~8-15s. Our pipeline: kuro-eye analysis (~3s) + FLUX rendering (~20-40s) + evaluation (~3s) + possible re-render = **26-86s worst case**.

Severity: HIGH — Users will perceive this as unacceptably slow compared to cloud alternatives.

RT-02: NO NATIVE TOKEN-SPACE REASONING

Attack: Gemini reasons about images in the same embedding space as text. Our kuro-eye can describe and analyze images, but it can't "think in images" — it converts visual information to text, losing spatial nuance. The feedback loop is text-mediated, not visually native.

Severity: MEDIUM — Affects complex spatial reasoning prompts.

RT-03: TEXT RENDERING GAP

Attack: FLUX.1-dev's text rendering is good by diffusion standards but nowhere near Nano Banana Pro's, which uses the LLM backbone to guarantee spelling. FLUX will still garble complex text, especially multi-word phrases and small font sizes.

Severity: HIGH — Text-heavy use cases (posters, infographics) will be noticeably worse.

RT-04: VRAM CONTENTION

Attack: With ~27GB loaded, there's only 5GB headroom. Loading a ControlNet adapter (~2GB), a LoRA (~200MB), and handling a large batch of inference steps could cause OOM. If kuro-main (the chat brain) is also loaded via Ollama, VRAM is oversubscribed.

Severity: MEDIUM — Crashes under concurrent load.

RT-05: FEEDBACK LOOP HALLUCINATION

Attack: kuro-eye's quality evaluation is subjective. A 7B VLM might rate a terrible image as "good" or a good image as "bad," triggering unnecessary re-renders or passing through failures. The evaluation criteria are fuzzy.

Severity: MEDIUM — Wasted compute on false positives/negatives.

RT-06: NO REAL-TIME GROUNDING

Attack: Nano Banana Pro can call Google Search to ground images in real-time data (weather, news, stock charts). Our pipeline has no web access during generation unless we explicitly add it.

Severity: LOW — Nice-to-have, not core.

RT-07: MODEL UPDATE DEBT

Attack: FLUX.1-dev and Qwen2.5-VL are both improving rapidly. FLUX.2 (32B) is already out but too large for concurrent loading. Qwen3-VL will likely drop soon. You'll be constantly chasing model upgrades.

Severity: LOW — Manageable with modular architecture.

RT-08: DISK CONSTRAINT

Attack: 200GB disk is tight. FLUX.1-dev base model (~24GB), Qwen2.5-VL 7B (~5GB), kuro-main (~5GB), plus LoRAs, cached latents, generated images, OS overhead. You'll hit 80% utilization fast.

Severity: MEDIUM — Needs aggressive storage management.

SECTION 06

Post-RT Mitigations & Refined Design

MIT-01: LATENCY → STREAMING UX + ASYNC PIPELINE

Solution: Don't hide the latency — *weaponize it*. Stream the thinking process to the user exactly like Nano Banana Pro's "Thinking mode" UX:

- Phase 1 (0-3s): Show kuro-eye's scene analysis in ThinkingEngine.jsx — "Analyzing composition... planning lighting... enhancing prompt..."
- Phase 2 (3-30s): Show FLUX denoising progress as a live preview (partially denoised images updating every 5 steps)
- Phase 3 (30-35s): Show evaluation — "Checking prompt adherence... Score: 8.5/10 ✓"

This transforms a 30s wait into an engaging "AI thinking" experience. Users perceive it as thorough, not slow. **Additionally:** Skip the feedback loop for simple prompts (kuro-eye confidence > 0.85) to cut latency to ~25s for straightforward requests.

MIT-02: TOKEN-SPACE GAP → STRUCTURED SCENE GRAPHS

Solution: Instead of free-text prompt enhancement, have kuro-eye output a structured JSON scene graph with explicit spatial coordinates, object relationships, and z-ordering. This compensates for the lack of native visual token reasoning by making spatial information explicit and machine-parseable. The orchestrator translates the scene graph into ControlNet conditions (depth maps, edge maps) + optimized prompt tokens.

MIT-03: TEXT RENDERING → COMPOSITE PIPELINE

Solution: For text-heavy requests, split rendering into two passes:

1. **Background pass:** FLUX generates the scene WITHOUT text (prompt instructs "leave space for text at [coordinates]")
2. **Text overlay pass:** kuro-eye specifies exact text, font, size, color, position. A Pillow/Cairo compositor renders pixel-perfect text over the FLUX output. This **guarantees** perfect spelling — the LLM handles language, the renderer handles pixels, neither does both.

MIT-04: VRAM CONTENTION → EXCLUSIVE MODE + OFFLOAD

Solution: Implement VRAM mutex in the orchestrator. When KURO::VISION is active, kuro-main (chat brain) offloads to CPU. Ollama supports this natively with

`OLLAMA_MAX_LOADED_MODELS` . Set kuro-eye and kuro-gen as priority GPU residents; kuro-main falls back to CPU inference (slower but functional). After image generation completes, kuro-main reloads to GPU. Additionally: run FLUX inference with `enable_model_cpu_offload()` from diffusers — this keeps only the active transformer block on GPU, reducing peak VRAM by ~30%.

MIT-05: FEEDBACK HALLUCINATION → CALIBRATED SCORING

Solution: Replace subjective scoring with specific, binary checks: "Does the image contain [object X]? Yes/No." "Is the text legible? Yes/No." "Is the lighting coming from the correct direction? Yes/No." Binary checks are far more reliable from a 7B VLM than numerical scores. Only trigger re-render on binary failures, not subjective quality dips. Additionally: provide kuro-eye with the enhanced prompt (not original) so it evaluates against what was actually requested.

MIT-06: GROUNDING → TOOL INTEGRATION

Solution: Add a pre-generation tool-use step where kuro-eye can request web data via your existing backend. If the prompt references "current weather" or "latest [X]", the orchestrator fetches data, formats it, and injects it into the generation context. This mirrors Gemini's Google Search grounding exactly — it's just tool use.

MIT-07: MODEL UPDATES → ABSTRACTION LAYER

Solution: The orchestrator talks to models through a standard interface (Ollama for VLMs, diffusers API for image gen). Swapping FLUX.1-dev for FLUX.2-dev (when it fits) or Qwen2.5-VL for Qwen3-VL is a config change, not a rewrite. Keep model selection in environment variables.

MIT-08: DISK → AGGRESSIVE PRUNING + EXTERNAL STORAGE

Solution: Store generated images with automatic 30-day TTL. Keep only 3 most recent LoRAs loaded; others on S3/Backblaze B2. Use symlinks for model directories so you can mount external block storage if TensorDock offers it. Run `ollama prune` to clean unused model layers.

Implementation Plan

Phase 1: Foundation (Day 1-2)

SERVER BOOTSTRAP + MODEL INSTALL

SSH into TensorDock VM, install CUDA drivers, Ollama, Python env with diffusers + torch, pull models.

Deliverables: kuro-eye (Qwen2.5-VL 7B) responding via Ollama, kuro-gen (FLUX.1-dev) generating test images via Python script, both coexisting in 32GB VRAM.

Phase 2: Orchestrator (Day 3-5)

KURO::VISION MIDDLEWARE

Build the orchestration layer as a new module in server.cjs. Image intent detection in semantic_router.js. Prompt enhancement chain. VRAM management. Feedback loop with binary checks.

Deliverables: POST /api/vision/generate endpoint. SSE streaming with thinking phases. Session memory for conversational editing.

Phase 3: Frontend (Day 6-7)

KUROCHATAPP.JSX INTEGRATION

Image generation UI in chat. Live denoising preview. ThinkingEngine.jsx phases for vision pipeline. Inline image display with edit controls.

Deliverables: Users type prompts in chat → see thinking → see image materialize → can request edits conversationally.

Phase 4: Quality (Day 8-10)

REFINEMENT + LORA TRAINING

Text composite pipeline for typography. ControlNet integration for spatial control. Train domain-specific LoRAs (KURO UI style, etc.). Real-ESRGAN upscaler for 2K+ output.

Deliverables: Production-quality image generation matching Nano Banana Pro in your target domains. Text rendering that's actually readable.

Estimated Total Cost

ITEM	COST
TensorDock RTX 5000 Ada VM	~\$0.60-1.20/hr (varies by config)
Models (all open-source)	\$0
LoRA training compute	~\$0 (on your own GPU)
External storage (optional)	~\$5/mo for B2
vs. Nano Banana Pro API	\$0.15/image (adds up fast)

BOTTOM LINE

KURO::VISION won't replicate Nano Banana Pro's unified architecture — that requires Google-scale training. But it replicates the **user experience**: type a prompt, watch it think, get a reasoned image, edit conversationally. In your specific domains (KURO UI assets, AgTech imagery, branded content), a fine-tuned FLUX + reasoning VLM pipeline will **exceed** Nano Banana Pro because it's specialized. The sovereignty guarantee — zero data leaving your server, no API costs, no rate limits, no content policy censorship — is the moat Google can never compete with.

