Autonomous Agents for Heterogeneous Data Integration and Local NLP: Experiences with Cloud and Edge Deployments:

Abstract

AI is changing quickly, and a big part of that is the rise of “autonomous agents” - essentially systems which can think for themselves, use different programs, and make choices on their own. This paper looks at two different projects using these kinds of agents for both bringing data together and for natural language processing (NLP). The first uses xAI’s Grok-3-mini, a large language model (LLM) in the cloud, to pull in all sorts of information from lots of different online stores at the same time. It’s set up with a ‘master and worker’ system for getting the information (scraping) without waiting for everything to finish in order, for matching items from different sites without needing to be specifically told what to look for (zero-shot entity alignment), and for keeping the data current and quick to access (using caching). The second project is about using LLMs on the device itself (on the ‘edge’) with Ollama and a ReAct loop to do a reverse image search. This is about being private, working even with no internet connection, and being able to handle multiple languages, and specifically to summarise information in Uzbek.

These two projects are quite different in how much they depend on the cloud or on being on your own equipment. In the online store data gathering, Grok’s ability to use other tools allows it to improve what it asks and to get info from elsewhere, and a clever trick with a time-limited cache in Redis makes things much faster. When compared to other ways of doing it, this method is a lot quicker and matches things much more accurately. But the reverse image search agent, with Ollama’s llama3.1, decides on its own which search engines to use (TinEye, Yandex, Bing), then looks at the picture on your device to come up with a description of it - and none of that picture is sent anywhere, so you remain private.

From actually building these, a few important things about designing agents become clear: how to get both the ability to handle a lot of work and the ability to work on its own, how important those 'thinking' loops are for deciding what to do, and how to adjust things for languages that don’t have a lot of online resources. And from my own experience learning by doing, it's essential to build things that can still operate if parts of them break, and to consider a mix of cloud and local processing. This work fits with the current exploration of AI that isn’t in one central place, and offers real-world advice for improving these agent systems when they have to work with all kinds of different data.

Introduction

I'm drawn to agentic AI because of its power to completely change how we deal with complicated, ever-changing information. Standard AI systems tend to be stuck in fixed sequences, but autonomous agents are flexible: they can think things through, use various tools, and change their plans as needed. This interest started when I was an undergrad in computer science and was playing with machine learning for processing data. It didn't take long, though, to see that applying this to the real world—think of online shopping, or analyzing pictures—requires systems that can cope with uncertainty, pull together information from many places, and work with very little help from people. Agentic AI, and specifically using large language models (LLMs), solves these problems by working like a person solving a problem: looking at things, thinking, doing something, and then improving the process.

The fact that things can be done in the cloud or directly on a device (at the "edge") was a big factor behind these projects. Cloud LLMs, like xAI's Grok-3-mini, have huge amounts of knowledge, can connect to things on the internet in real time, and can easily handle lots of requests. They're perfect for things where you need up-to-the-minute info from the web—like keeping an eye on the market; the agents can adjust what they're asking for and get new updates as they go. Edge-based models, though, using things like Ollama, are about protecting your privacy, getting answers quickly even without an internet connection, and tailoring the system to specific languages or areas of expertise. This is incredibly important for something like a reverse image search where someone might be using private, personal photos and doesn't want to send them to a server somewhere.

In Uzbekistan, where Uzum, Asaxiy and Olcha are the main shopping platforms but all the data is separated and in different places, getting all the information together is absolutely essential. My first project does this by building a system of interacting autonomous agents that collects, organizes, and analyzes market data as it happens. Using Grok, it does 'zero-shot entity alignment' - meaning it classifies products as EXACT, CLOSE, ACCESSORY, or UNRELATED without being trained beforehand. This makes comparing prices much easier and also shows how cloud agents can make the economy more accessible in countries that are still developing.

Moving to edge computing, the second project tackles worries about privacy when dealing with images. Tools like Google Reverse Image Search worldwide can leak your data, and this is especially true in places with a bad internet connection or strict rules. I've used Ollama locally, with a 'Reason-Act' cycle, so the agent can independently pick and link search tools (TinEye, Yandex, Bing), create descriptions in many languages (including Uzbek), and run entirely offline once it's set up. This fits with the increasing demand for AI that's spread out, where calculations are done on your device to lessen how much you rely on central computers in the cloud.

These experiences tie in directly with the research at Mohamed bin Zayed University of Artificial Intelligence (MBZUAI). MBZUAI's work on agentic systems puts a lot of focus on agents working together, as you can see in their published work on decentralized learning and "swarm intelligence". Their explorations of fine-tuning local models are similar to how I've adapted Ollama for Uzbek NLP, tailoring the prompts for summaries that make sense in the culture. And their research on spreading LLMs out, including edge AI for devices with limited power, mirrors my own combined strategy of using the cloud for large amounts of data and the edge for private information. Plus, their focus on agents that deal with multiple types of information inspires my future plans to put together vision and language models.

I'm presenting these projects in this paper as much as a record of how I learned as a description of the technical stuff. While building the shopping comparison tool, I discovered how valuable asynchronous architectures are when dealing with how quickly real-time data can change. And the reverse image search agent showed me how 'reasoning loops' really create independence, and cut down on needing lots of pre-programmed rules. Both of them have pushed me to pursue further study at MBZUAI, where I want to add to the development of agentic AI for languages that aren't widely supported and distributed systems.

Beyond simply how well they work, these projects are about my own development. My first attempts at collecting data from websites showed me that doing things one after another (synchronously) wasn’o very efficient, which led me to asyncio and master-worker setups. Experimenting with the tool-calling of Grok made me understand how powerful refining your search terms is when you’re looking for something that’s not very clear. And integrating Ollama locally showed me the difficulties of shrinking models for use on edge devices, and how important it is to be able to run them quickly. From this I’ve learned that agentic AI isn’t just about getting a good score on a test; it’s about making systems that adapt to the user’s situation—whether it’s a busy online store or someone looking at their own photos.

Because of how things are with digital technology in Uzbekistan - where the internet is variable and local languages like Uzbek are often overlooked by global AI - these agents fill in the gaps. Combining shopping data gives people a single view, while local NLP makes sure that tools are available for everyone. And in line with MBZUAI’s goals, this work shows how agentic AI can help to create fairer technology.

Project 1: Heterogeneous E-commerce Data Integration via Autonomous Agents

In Uzbekistan and similar developing places, shopping online is all over the place. Customers really have a hard time figuring out which is best when looking at things on Uzum.uz, Asaxiy.uz, Olcha.uz and others. That’s because these sites are built in all sorts of ways - using React Single Page Applications, JavaScript for building pages on the server, or Nuxt.js/Vue - and as a result, the information, how prices are shown, and descriptions of products aren’t the same from site to site. The biggest difficulty is getting all this different information together: pulling in what’s for sale right now, matching up the same items when there’s no one to tell the system what’s what, and then offering sensible suggestions. Old-fashioned ‘Extract, Transform, Load’ systems won't work because they are too fixed and can’t deal with websites that are always changing or with product details in multiple languages.

To solve this, I've built a self-operating team of 'agents' and it's all powered by xAI’s Grok-3-mini. It’s designed with a 'master and workers' system, and with Python's asyncio, everything happens at the same time. The main agent (in the file agents/master.py) is in charge of everything: it understands what you’re asking for, looks in a quick-access store of info, tells the scraping agents to get to work, figures out which items on different sites are actually the same, and then sorts the results for you. The scraping agents (in agents/worker.py) are each set up for a specific marketplace and use Playwright to control a Chrome browser for getting the information.

As an example, in master.py, the overall control uses asyncio.gather() to get all the scraping agents going at the same time:

```
async def orchestrate(self, query: str) -> Dict:
    if await self.cache.check(query):
        return await self.cache.get(query)
    workers = [UzumWorker(), AsaxiyWorker(), OlchaWorker()]
    raw_listings = await asyncio.gather(*[w.scrape(query) for w in
workers])
    aligned = await self.reasoning.align(raw_listings, query)
    ranked = self.rank(aligned)
    await self.cache.set(query, ranked)
    return ranked
```

Because of the way it's built, the scraping happens without one step having to finish before the next can start. Each 'worker' is looking after getting around the site, pulling out the details of each item, and getting those details into a standard format all on its own. The programs that actually do the scraping (and they're in the tools/ directory) are built using a basic blueprint (tools/base.py) and absolutely must have asynchronous functions for going to a page (navigate()) and getting the list of items (extract_listings()). And for Uzum (in tools/uzum.py) in particular, the program pauses until the React Javascript on the page is fully finished loading.

```
async def navigate(self, query: str):
    await self.page.goto(f"https://uzum.uz/search?q={query}")
    await self.page.wait_for_selector(".product-card")
```

Asaxiy (the asaxiy.py tool) works through SSR content and uses Javascript to calculate prices which change. Olcha (olcha.py) is in charge of Vue components. The basic information it gets is then put into a standard format by Pydantic models (in core/models.py) which are very specific about what type of data each piece is - for example, a ProductListing has a title, a price in Uzbek Som (price_uzs), and a web address (url).

Importantly, Grok is key to making searches clearer and getting information from outside the system. Specifically, Grok, in core/reasoning.py, uses its tools to clarify what you mean when your search isn't quite specific enough.

```
async def refine_query(self, query: str) -> str:
    response = await grok_api.call_tool("refine", {"input": query})
    return response["refined_query"]
```

When doing zero-shot entity alignment, Grok looks at listings in a neat JSON format and looks at the brand, model, specifications and titles in Uzbek, Russian and English. It sorts these into four groups: EXACT (a perfect match), CLOSE (a very similar match, for instance, if it's just a different colour), ACCESSORY (something that goes with the item), and UNRELATED. If the large language model isn't working, Grok has some simpler methods in reasoning.py using keywords to help.

The caching system (in core/cache.py) uses Redis and aioredis for quick work, and things in the cache disappear after a default of 3600 seconds. It uses SHA-256 hashes of your search to make the cache keys, and stores the listings that have already been aligned as the value. If it finds a search in the cache, it skips looking for it again on websites; if it doesn't, it updates things after it's done. Without the cache, each search took 15 to 20 seconds, but with it, repeated searches are under a second, a 95% improvement. The cache doesn't stay forever, though, as it will expire after the TTL, causing Grok to look for things again on frequently changing websites.

The system works well: for a search for "iPhone 14 128GB", it brought together over 50 listings, identified 20 of those as being an EXACT or CLOSE match, and then sorted them by price and relevance. When people checked the results against what they knew to be correct, the system was correct 92% of the time. The bilingual Streamlit interface (ui/app.py) displays the results and the steps in Grok's thinking, so you can see how it arrived at the answer.

You can find the code on GitHub at https://github.com/uzbtrust/Heterogeneous-Market-Data-Integration.

I really got to grips with building systems that can handle things going wrong during this project. The way I handled errors (core/exceptions.py) with a structure of ScraperException and ReasoningException and so on, showed me how to pass on error messages in a reliable way. And I found that a self-governing system does best when it's broken into sections - keeping the scraping, thinking and caching all separate allowed me to improve each bit little by little. Looking at the changes I made to the code, the earliest ones (February 11th, 2026) were just about getting the basic scraping working. Then I added caching on February 18th, and I improved things further on March 4th. That progression actually shows how my understanding of building good, production-ready systems improved too.

Project 2: ReverseLens – Autonomous Reverse Image Search Agent with Local NLP

Figuring out what's in a picture, being sure something is what it claims to be, or locating similar images is hugely helped by reverse image searching. However, most businesses doing this require you to send your picture to their computers on the internet, and that's a problem for your privacy. When you need to be careful about privacy, or you are looking at personal photos in places where the internet isn't great, you really need a program that works on your own computer. The tricky bit is making a program that can look at images when not connected to the internet, decide how to find information, and then write descriptions of the image in many languages, and importantly, in languages that aren't used very much on the internet at all - like Uzbek.

ReverseLens solves this by using a ReAct (Reason-Act) system which is driven by the llama3.1 model from Ollama, and it does everything on your device. The design is about running on the 'edge' - your image is prepared, the program thinks about it, and the outcomes are saved, all without needing anything from the internet once it's installed. The ReAct system (in the file called services/react_agent.

```
async def react_loop(self, image_path: str) -> str:
    thoughts = []
    while not self.done:
        thought = await ollama.think(image_path, thoughts)
        action = await ollama.call_tool(thought)
        observation = await self.execute(action)
        thoughts.append({"thought": thought, "action": action, "obs":
observation})
    return await ollama.summarize(thoughts, lang="uzbek")
```

The `search_tineye`, `search_yandex`, and `search_bing` functions (from PicImageSearch) are all found in the file `services/tools.py`. The agent decides which of these to use by thinking about it, it doesn't just go through them in a set order. For example, if TinEye doesn't find many images, it could then use Yandex to look over a much wider range of material.

As for `services/search.py`, the things in that file are responsible for dealing with uploading images and interpreting the results.

```
def search_tineye(image_path: str) -> List[Dict]:
    searcher = TinEye()
    results = searcher.search(image_path)
    return [{"url": r.url, "desc": r.desc} for r in results]
```

We used Ollama because it's open to all and runs AI processing on your own computer or phone. This makes sure your information is private as it doesn't use any cloud based services; you can run it without internet access either in a Docker container or directly and your data stays with you. To get it to work with Uzbek, the instructions you give it to summarize something are adjusted to say "Summarize in Uzbek, and importantly, think about the cultural background." This multilingual natural language processing can deal with descriptions in Uzbek, Russian, or English.

The image processing part (in the file called services/preprocess.py) uses Pillow to make images a good size, clearer and with better differences between light and dark, so finding what you're looking for is more accurate. These results are saved in a temporary store (in utils/cache.py - using a JSON format with MD5 security) and then permanently in a SQLite database (utils/db.py) to show a history of what's been searched for. And you need a user login (done with JWT authentication in utils/auth.py) to see only your results.

What it does by itself is quite a thing. When given a picture of a famous place, it first used TinEye (and found only one match) but realized that wasn' and then switched to Yandex (finding ten matches) and gave this answer: "This picture shows the Amir Temur statue in Tashkent, with modern buildings in the background." On a computer it took on average 5 to 10 seconds to do this, but if the answer had been in the temporary store, it was under a second. When we checked about privacy, it showed that no information left your computer after you had set it up.

Here's the link to the code on GitHub: https://github.com/uzbtrust/ReverseLens.

From this, I really understood how useful a 'thinking in a loop' system is for giving an AI true independence. The first versions of the code (from February 26th 2026) just linked to TinEye, then on March 4th I reworked it to use the ReAct method. I found I had to find a balance between what the computer had and what it needed, and reducing the size of llama3.1 by 40% reduced how much memory it used.

General Analysis and Comparison

At the heart of both these projects are abilities you'd expect from an intelligent helper: they think things through to decide what to do and use different tools to do things. In the case of the e-commerce assistant, Grok's thinking sharpens your requests and makes sure it understands what things are. With ReverseLens, Ollama's ReAct system decides which tools to use and in what order. And the ability to use tools is the really important part – Grok goes out to the internet for information, while Ollama starts searches on your own computer. Both of them run into the same kinds of problems, specifically when something is unclear (like when product titles or pictures are in multiple languages) and also making sure they don't break down, using things like error messages and ways to do something else if the first approach doesn't work.

The way they're put to use is where they are different. The first project depends on the cloud (using the Grok API) for being current and being able to handle lots of use, so you can get results as things are happening. But that means a little delay and it will cost money.

Latency	**Project 1 (Cloud/Grok)**	**Project 2 (Local/ Ollama)**	**Trade-off**
Latency	5-15s (scraping + API); <1s cached	5-10s (local inference); <1s cached	Cloud: Network-dependents; Local: Hardware-bound
Privacy	Moderate (API calls expose queries)	High (fully offline post-setup)	Cloud: Data sharing risk; Local: No external leaks
Cost	API usage fees; free tier limits	Zero (open-source, local run)	Cloud: Scalable but billable; Local: Upfront hardware
Data Freshness	High (real-time scraping)	Medium (cached results, no live updates)	Cloud: Dynamic; Local: Static unless manual refresh
Scalability	High (cloud reources)	Low (device-limited)	Cloud: Elastic; Local: Fixed capacity
Language Support	Broad (Grok's multilingual)	Customized (Uzbek prompts)	Cloud: Global; Local: Fine-tunable for niches
Autonomy Level	High (query refinement, alignment)	High (ReAct decides tools)	Both enable iteration, but cloud integrate externals

When I was building the agents themselves, I really got how vital it is to make things in separate pieces, particularly keeping the “brain” (the main controlling process, like ReAct) distinct from the actual tools they use. That makes everything much more flexible and you can use parts again and again. And I figured out the balancing act with these hybrid approaches to systems, using the cloud for having lots of different options, but then doing the serious, detailed work on your own computer.

Future of Agents and Plans

The way AI is going, with these “agentic” systems, is definitely towards a mix of cloud computing and doing things right on the device (at the “edge”), handling both images and text, and especially, using AI directly on all those internet-connected “things” (IoT). A combined approach of this sort would allow agents to split up their work: the cloud would be for big, complex calculations, and the device itself would make choices immediately, which is faster and keeps your information more private.

I’m aiming to do research at MBZUAI on adjusting existing AI models to work specifically with the Uzbek language. I’m going to start with versions of Ollama. This might mean collecting data for languages that don’t have much of it, and perhaps even making large language models that are spread across many locations to safeguard and celebrate culture.

Conclusion

These projects are pretty much a story of my time working with AI agents, going from plugging into cloud services to using language processing on my computer. They’ve shown me I’m a good match for what MBZUAI is doing with systems that aren’t all in one place.