ChatGPT

Dynamic Multi-Tenant ML Inference Integration on AKS

Introduction and Context

The company’s current system processes machine learning (ML) inference tasks using a **Python application on Azure Kubernetes Service (AKS)** with a classic Celery setup. The architecture comprises a **REST API** (fixed-scale deployment) that enqueues jobs into **Redis** (acting as a message broker), and a pool of **Celery worker pods** (auto-scaled via KEDA) that consume tasks from the queue. All workers currently use a single “standard” ML engine for inference. This design (illustrated in **Figure 1**) decouples request handling from long-running ML computations, enabling asynchronous processing and auto-scaling of workers based on load.

Figure 1: Current architecture. Client requests hit the REST API, which enqueues tasks in Redis. Celery workers fetch tasks and execute them using a fixed ML engine, storing results for retrieval.

With growing demand for customization, **new requirements** have emerged:

- **Pluggable ML Engines:** Each client should be able to develop and deploy their own custom ML inference engine (e.g. a model or algorithm), packaged either as a container or a code module. All custom engines must conform to a standard interface (so the system can invoke them uniformly).
- **Dynamic Task Routing:** When a task is enqueued, the message will specify which engine to use. The system must dynamically dispatch the task to the appropriate client-provided engine at runtime.
- **Zero Impact on Core Deployment:** Adding a new engine should **not require any changes to the core application’s code or a redeployment** of the main system. In other words, the platform should discover and integrate new engines on the fly.
- **Secure Multi-Tenancy:** Each client’s engine must be isolated to prevent any possibility of one client’s code or data interfering with others’. Executing **untrusted code** safely is a paramount concern.

This report evaluates several approaches to achieve these goals, compares them on **scalability, security, maintainability, and integration ease**, and then recommends an optimal solution. We also provide an implementation guide (code snippets, deployment YAMLs) for the chosen design, along with security best practices like sandboxing and container isolation.

Approaches for Integrating Custom Engines

We consider four broad strategies for integrating client-provided ML engines into the AKS-based application:

1. **Dynamic Module Loading (In-Process Integration)**
2. **Plugin Architecture (Extensible via Python Plugins)**
3. **Containerized Engines (External container services or sidecars)**

4. **Kubernetes-Native Orchestration (Replacing Celery with K8s Jobs or similar)**

Each approach is detailed below with its mechanism, pros, and cons.

1. Dynamic Module Loading at Runtime

Dynamic module loading means the worker process can load and execute new Python code (modules) at runtime, without those modules being present at application start. In this scenario, clients would provide their ML inference code (e.g. a Python `.py` file or package) implementing a predefined interface (such as a function `predict(data)` or a class with a known method). The system could be designed to load such modules on demand when a task specifying that engine arrives.

Implementation: One could maintain a directory (or a package index) where clients' modules are placed. When a task with `engine_name="clientX_engine"` comes in, the worker uses Python's import mechanisms (`importlib`) to load the module dynamically and then invoke the standard entry point (e.g. `module.predict(request_data)`). For example:

```python
import importlib

def run_inference(engine_name, data):
    engine_module = importlib.import_module(f"engines.{engine_name}")
    result = engine_module.predict(data)  # Standard interface call
    return result
```

This code would live in the Celery task execution path. The dynamic import allows new modules (engines) to be picked up if they're present in the `engines` package directory without changing the worker code.

Pros: This approach keeps everything in-process. It's simple in concept and doesn't introduce extra services. Calls to the custom engine are just Python function calls (no network overhead). If the core environment can be made aware of new modules (e.g. by mounting volumes or updating PYTHONPATH), then adding an engine might be as easy as dropping a file in the right place.

Cons: Security and isolation are major issues. Running untrusted client code inside the main worker process is risky – a buggy or malicious engine could crash the worker or compromise the whole application. Python has no robust sandbox mechanism for untrusted code; "in-process sandboxing" is not what Python was designed for [1]. This means a malicious plugin could potentially access global variables, read/write files, or even make network calls if not very carefully restricted. Techniques like Python's `exec` or import hooks can load code, but **preventing that code from performing unauthorized actions is extremely difficult**. A Stack Exchange discussion on safe plugin architectures concluded that if you need to execute untrusted code, you essentially "want the benefits of container technology" – i.e. to isolate it at the OS level [2].

Maintainability is also a concern: because all code runs in one process space, a misbehaving engine (e.g. one that leaks memory or changes global state) can affect subsequent tasks. It's recommended to restart the worker process after each task when running untrusted code to avoid cross-contamination [3], but that would hurt performance and complexity. Scaling is limited to scaling the whole worker process – you can't independently scale one client's engine; they all run under the same worker instances.

Summary: Dynamic loading achieves the goal of no core code changes per new engine (if implemented generally), but it fails on security and possibly stability. This approach would only be viable if all client code is fully trusted (which defeats the "untrusted code" requirement) or if additional sandbox measures (like running the code in a restricted virtual machine or using a language with sandboxing) were used – which is beyond Python's native capability.

2. Plugin Architecture (Extensible via Plugins)

A plugin architecture is similar to dynamic loading but provides more structure. Typically, the main application defines a **plugin interface or abstract base class** that all engines must implement. Clients then package their engine as a **plugin module** (potentially an installable Python package or wheel). The system loads plugins either by scanning a folder or via entry points (if using Python packaging).

For instance, one could use a plugin framework or a pattern like:

- Define an abstract class `MLEngine` with a method `predict(data)`.
- Clients implement `MyEngine(MLEngine)` in their module.
- At startup or on demand, the system discovers all subclasses (e.g. via `pkg_resources.iter_entry_points('ml_engines')` or by scanning a known plugin directory).
- Each plugin is registered under a name (perhaps defined in its metadata), so that a task can reference the engine by name.

Pros: Compared to arbitrary dynamic code, plugins enforce a standard interface, which improves maintainability. The discovery of new plugins can be automated. If using Python's packaging entry points, a client could distribute their engine as a Python package that the admin installs into the environment (though that does violate the "no deployment change" rule unless done dynamically). Structurally, it's cleaner and can prevent some errors (an engine that doesn't implement the required interface could be rejected).

However, **the fundamental drawbacks regarding security remain** – the plugin code still runs in the same process. A plugin can call any Python code the process can, possibly interfering with others. As one expert noted, Python lacks a built-in security sandbox, so the only robust isolation is to run untrusted plugins in separate processes or sandboxes [1]. In effect, a plugin architecture by itself doesn't solve the safety problem; it's just a formalized version of dynamic loading.

Scaling and independence: All plugins share the same worker process resources. You cannot isolate resource usage per client (e.g. one client's heavy model could consume all CPU of a worker). Also, if a plugin needs specific Python dependencies, those have to be installed in the main environment, potentially causing dependency conflicts between plugins.

Summary: A plugin system improves organization and enforces a contract for engine developers, but on its own it doesn't meet the isolation requirement. It might be acceptable in a controlled extension scenario (trusted internal plugins), but for **multi-tenant untrusted extensions, in-process plugins are too risky**.

Industry practice for SaaS extensibility is to avoid running user code on your main server directly – either push it to the client side or isolate it on separate infrastructure [4] .

3. Containerized Engine Deployment

This approach treats each custom ML engine as an independent service, running in its **own container(s)**, possibly as a separate deployment on the AKS cluster. Instead of loading client code into the main application process, the main system will **invoke the engine via inter-process communication** – typically an HTTP/gRPC call to the engine's service, or by publishing a message that that engine's service consumes.

There are two variations considered here:

- **3A. Sidecar containers:** The engine container is run as a sidecar in the same pod as the worker.
- **3B. External service pods:** The engine is deployed as one or more separate pods (with its own Service endpoint) independent of the main worker pod.

Sidecar Approach: In a sidecar model, whenever a Celery worker pod starts, it includes, say, a secondary container running the client's inference code (perhaps an API server or a small service that loads the client model). The worker can communicate with the sidecar via localhost networking or IPC. Sidecars are typically used to provide supportive features (like logging agents or proxies) to a primary app; here, one could consider the engine as a co-process that the worker calls for inference.

- **Pros:** Communication is local (within the pod), so latency is low. Isolation is improved compared to in-process – the engine is a separate process, so memory isn't shared. If the engine crashes, it might only affect its container.
- **Cons:** The sidecar model becomes unwieldy when there are many different engines. A given pod's composition is fixed at deployment time – you would need to define a pod template per engine or per client. That means adding a new engine necessitates deploying a new worker+sidecar configuration (contravening the "no core deployment change" rule). If you try to include **multiple sidecars for multiple engines in one pod**, that pod grows in complexity and resource usage (running all engines even if only one is needed at a time). It's also not truly dynamic; Kubernetes doesn't support adding a new container to existing pods on the fly (except ephemeral containers for debugging, which is not a solution here). In summary, **sidecars don't scale well for this use-case** – they're better when the sidecar logic is the same for all pods (e.g. a logging agent), not when each client requires a different sidecar.

Given these drawbacks, we lean toward **external service pods (3B)** as the containerization strategy:

External Service Approach: Each custom engine is packaged as a container image (for example, implementing a small web server that listens for inference requests on a certain port). When a client onboards a new engine, they deploy this container to the AKS cluster (or the platform's operators deploy it on their behalf) as a standalone Deployment (and corresponding Service for networking). The main application does **not** incorporate the engine code at all – it will interact with it over the network.

There are multiple ways to invoke the engine service:

- The Celery worker task, upon receiving a job for engine "X", makes an HTTP request to `http://engine-X.my-namespace.svc.cluster.local/predict` (for example) with the payload. It waits for the response and then continues the workflow (e.g. saving result to DB or returning via the API).
- Alternatively, use messaging: the task could publish the request to a dedicated queue (say Redis or Azure Service Bus) that the engine service listens on. In effect, each engine service could have its own mini task queue and worker. (This starts to resemble having multiple Celery apps, one per engine.)
- A more radical approach: have no central Celery at all, and instead route tasks directly to engine services (more on this in Approach 4).

Pros: This model offers **strong isolation** and clear multi-tenancy separation. Each engine runs in a separate container which can be constrained with CPU/memory limits and OS-level security (Linux user permissions, seccomp, AppArmor, etc.). If one engine crashes or leaks resources, it doesn't directly impact others or the core application. It also allows **polyglot** implementations – a client could write their engine in R or Java or any language as long as they wrap it in the agreed interface (e.g. a REST API). The core system doesn't care; it just calls the service.

Scalability is improved: you can scale each engine deployment independently. For example, if Client A's engine is very popular, the cluster can run many replicas of the engine-A service behind a load balancer, without scaling the core workers that handle other tasks. Conversely, idle engines consume no core resources (their service could even be scaled to zero if using something like Knative for auto-scaling to zero on no traffic). This maps well to multi-tenant model serving best practices, where tenant-specific models are deployed separately for isolation [5] [6].

Maintainability: The core application code is simpler – it just needs a generic way to call engines. Each engine's codebase is separate, making it easier for clients to manage their own code without entangling with the main code. There's no need to install all possible model dependencies in the main app environment; each engine brings its own dependencies in its container. This avoids "dependency hell" and bloated images.

Crucially, adding a new engine does **not require redeploying the main app**. The client (or ops team) can deploy the new engine container, register it (e.g. inform the system of the engine's name or URL), and from then on any tasks labeled for that engine will be routed to it. The main system doesn't change; it just uses the engine identifier to lookup where to send the request.

Cons: There is additional **infrastructure complexity**. Instead of one homogeneous system, we now have potentially dozens of small microservices (one per client engine). This requires proper DevOps tooling for clients to build and publish their container, and for the platform to deploy it (possibly in a separate Kubernetes namespace for that client, to enforce isolation). An orchestration or discovery mechanism is needed so the core app knows how to reach each engine (this could be as simple as a naming convention or a lookup table mapping engine IDs to service DNS names).

Performance overhead: Calling an external service adds network latency and serialization overhead. Instead of a function call in memory, data must be sent over HTTP. For most ML tasks which are moderately heavy (tens of milliseconds or more), this overhead is usually acceptable. If the payloads are large (images, etc.), you might need to pass references (like an object store path) rather than raw bytes to avoid network overhead. There's also potential duplication of work – e.g., multiple engines might each have to load common libraries; but this is the price of isolation.

Another consideration is **result collection**. If the core system is making synchronous calls to the engine, it can get the result directly. But if it hands off the task asynchronously (e.g. via a message), the system needs to capture the engine's output. A simple pattern is that the Celery task (in the main app) calls the engine's API and waits for the reply (with a timeout). This is synchronous from the worker's perspective but still async relative to the user request (since the original API call returned immediately after enqueuing to Celery). The result can then be stored (e.g. in Redis or a database) for the user to fetch. This approach is straightforward but means a Celery worker is occupied during the engine computation (not doing other tasks). If that is a concern, a more asynchronous message passing could be used (engine pushes result to Redis, etc.), at cost of complexity.

Despite these complexities, the containerized service approach is widely regarded as a sound method for ML model serving in multi-tenant scenarios. In fact, many MLOps platforms embrace a similar idea: package the model and inference logic into a Docker image (a **model-serving runtime**), then deploy it to a **serving platform** that can autoscale and expose it as an API [7] [8]. Tools like KServe or Seldon Core follow this pattern – they create a Kubernetes Deployment for each model (or sometimes share models on one runtime) and handle routing. In our case, we are essentially building a lightweight version of that concept.

4. Kubernetes-Native Task Orchestration (Replacing Celery)

The final approach challenges the use of Celery/Redis altogether and asks: can we rely on Kubernetes or cloud-native mechanisms to dispatch tasks to engines dynamically? This could manifest as:

- **One Job per Task:** When a new inference task comes in, spawn a short-lived Kubernetes **Job** to handle it. The job would run the appropriate engine container (or run a wrapper that loads the engine). After processing one message, the job pod terminates.
- **Event-Driven Invocation:** Use an event system (e.g. KEDA's ScaledJob, or Argo Events & Workflows) that listens to the Redis queue (or any trigger) and launches tasks in containers accordingly. KEDA's ScaledJob CRD, for example, can watch a queue and **create one Job per message** [9]. In such a scenario, Celery's role is completely eliminated – the queue + KEDA (or similar controller) takes over the responsibility of scheduling containers for work.

Pros: This approach maximizes isolation – each task runs in a fresh container sandbox, fully isolated from others. It also means resource allocation is very granular (no multi-tenant code in one process at all). Scalability is potentially very high: in theory, each incoming request can spin up a new pod (if the cluster has capacity), allowing massive concurrency. This is how serverless batch jobs or systems like Azure Batch/AWS Batch or Kubernetes Jobs operate. Indeed, the Airflow project moved from Celery to a Kubernetes Executor for similar reasons – packaging each task as a pod gave better isolation and removed the need for a persistent worker process [10].

Crucially, this model adheres to not modifying the core app for new engines: the decision of which image to run could be encoded in the job spec dynamically. For example, a message might include an engine identifier which the job-launcher uses to select the corresponding container image for the Job. If using KEDA ScaledJob, you might define separate triggers per engine type, or one trigger that uses a generic worker image that then loads the engine (some creativity needed here, since ScaledJob uses a fixed template – perhaps a single “engine-runner” image that takes the engine name as an argument and then `docker pull` or chain-exec the actual engine container – though that becomes complex).

Cons: The overhead of starting a container for every single task can be significant. Even a lightweight container might take a few seconds to schedule and start on Kubernetes, especially if scaling from zero. For high-throughput or low-latency requirements, this overhead could be unacceptable. Celery, by contrast, keeps workers alive to process many tasks, amortizing startup cost. So this approach is best suited when tasks are large/long-running (so that a few seconds startup is negligible), or tasks are infrequent enough that on-demand spin-up is fine. KEDA documentation notes that using Jobs is preferable for **long-running executions**, whereas for many short events a Deployment-based scaler is better [9].

Another challenge is **state and result handling**: If each task is an independent pod, you need a mechanism to get the result back to the user. One method is to have the job write the result to a known location (e.g. update a database row or put a result message in another queue). The user might then poll for the result or be notified. This is similar to how we manage results with Celery (which can use a result backend or polling via task IDs). Implementing this from scratch requires careful design (or use an existing workflow engine).

Maintainability & complexity: Replacing Celery with K8s primitives can remove one layer of technology (no Celery, no Redis perhaps if using cloud events), but it pushes complexity into Kubernetes configurations (CRDs, controllers, etc.). Engineers may need to be comfortable debugging Kubernetes jobs and events. Additionally, logs and monitoring might become trickier – instead of a few worker pods, you have potentially thousands of short-lived pods; capturing their logs or failures requires robust tooling.

Security: The security is strong in terms of isolation between tasks. However, if multi-tenancy is at the cluster level (multiple clients’ jobs on one cluster), one must still ensure the usual container security (no broad cluster permissions in pods, network policies to restrict pod communication, etc.). It’s advisable to run untrusted code with a locked-down pod security context (non-root user, limited Linux capabilities) and possibly on dedicated nodes (to avoid side-channel interference).

Example: To illustrate, imagine using KEDA with Redis: when a message arrives, KEDA could spawn a Job using the **engine’s container image** to process it. The flow: **no idle workers** – if the queue is empty, nothing runs. If 10 jobs come in at once, 10 pods are created (subject to configured limits). Each Job pod pulls the specified image (assuming it’s available) and runs the inference, then exits. This model was suggested by a user as an alternative to Celery in a Kubernetes forum discussion, noting it can improve resilience under high load by avoiding persistent worker bottlenecks [11]. It’s essentially leveraging Kubernetes as the “task queue executor”.

Hybrid approach: It’s worth mentioning that these approaches can be combined. For instance, one could keep Celery for the task queue semantics but use it to launch engine-specific containers. A Celery task might do nothing but trigger a Kubernetes Job for the real work and then monitor it or fetch result. However, this hybrid might add unnecessary latency.

Given all considerations, **Approach 3B (Containerized External Services)** emerges as a strong candidate, possibly augmented with Kubernetes-native scaling features. Approach 4 (direct jobs) is very powerful for isolation but might be overkill unless we anticipate very large tasks or want to eliminate Celery entirely. Next, we compare the approaches on key criteria to solidify the choice.

Comparative Evaluation of Approaches

Let’s assess each approach against the core criteria:

- **Scalability:** ability to handle increasing load and many different engines efficiently.
- **Security:** isolation of untrusted code, preventing malicious or accidental interference.
- **Maintainability:** ease of managing the system (for both the platform team and client developers), including adding/updating engines.
- **Ease of Integration:** how simple it is for clients to integrate their engine without platform changes, and their deployment independence.

Dynamic Loading (In-Process):

- *Scalability:* Moderate. Celery already scales workers via KEDA, so more tasks => more worker pods. However, all engines share those workers. A heavy engine could consume a worker’s resources and starve others. There’s no fine-grained scaling per engine. Also, a large number of different engines might bloat the worker image (if each engine has different dependencies, installing all is untenable).
- *Security:* Poor. All code runs together. A malicious engine can access memory/state of others. Sandbox options are virtually non-existent in vanilla Python [1] . A buggy engine could crash the process. This fails the multi-tenancy isolation requirement.
- *Maintainability:* Poor to Moderate. It’s straightforward in code (load module, call function), but debugging issues is hard because if something goes wrong in the worker, it could be any engine’s fault. Upgrading or changing one engine could necessitate restarting all workers. Clients need to coordinate library versions with the main environment.
- *Client Integration:* Easy in theory (just supply a Python module), but in practice, delivering that to the running system is tricky. It might require clients to upload code to a shared volume or an admin to pip install their package. There’s no built-in deployment mechanism here, which could lead to manual steps (and thus potential downtime or redeploys – unless a live-reloading scheme is built).

Plugin Architecture (In-Process with Structure):

- *Scalability:* Similar to dynamic loading – limitations of one process space. No independent scaling of individual plugins.
- *Security:* Poor, same fundamental issue (in-process). Even if you authenticate the plugin package or enforce code reviews, a bug could still take down the service.
- *Maintainability:* Better structured – engines must implement interfaces, which reduces accidental misuse. Discovering plugins can be automated. However, dependency management and global state issues remain. Each plugin might need to be tested in context of all others since they share runtime.
- *Client Integration:* If using entry-point mechanism, a client could publish a pip package. But installing it into production without redeploy is non-trivial. You’d need a mechanism for dynamic installation (which itself is risky if done on the fly). Alternatively, all client plugins could be baked into the image periodically, but that violates the “no redeploy” wish. So integration is not seamless unless one builds a custom plugin loader system (with perhaps clients uploading code through an API, which is complex and still insecure).

Containerized Engines (External Services):

- *Scalability:* Excellent. Each engine can scale out independently with its own deployment. The cluster can handle different engines with different resource profiles (one model might require GPU and can be scheduled on GPU nodes, others CPU only, etc.). This is essentially horizontal scaling at the service level [12]. The main REST API and queue can remain fixed, and back-pressure can be managed by scaling the engine services or the Celery orchestrator. This model aligns with cloud-native scaling: "scale services independently without worrying about the underlying communication" [12]. The drawback is potential overhead of inter-service calls, but that is outweighed by the ability to cache models in memory across calls and not reload on each task (as would happen with jobs).
- *Security:* Very good. Container boundaries are an industry-standard security layer. While not perfect, they are *far* better than running untrusted code in-process. We can enforce network restrictions (each engine service only communicates via the API, and cannot directly reach the database or other engines if not needed), use Kubernetes Namespaces for tenancy, and apply Pod Security Policies. Even if an engine is compromised, the damage is largely contained to that engine's scope (it might try to abuse resources, but Kubernetes resource limits and isolation will cap that). As one StackExchange answer noted, isolating untrusted code in a separate process or container is the recommended solution – essentially *"fork it off in a chroot or jail, or run in a separate VM"* [2]. Docker containers are a form of such jails. This approach thus satisfies the multi-tenancy isolation requirement best.
- *Maintainability:* Moderate to Good. On one hand, having multiple services means more things to monitor (each could have its own logs, metrics). But modern tools (Prometheus, ELK, etc.) can aggregate that. Kubernetes makes deployments relatively uniform (each engine is just another Deployment + Service). The core application remains simpler (no need to handle loading errors etc., just network calls). Upgrading an engine or rolling it back is decoupled from others. One challenge is versioning – if the interface between core and engines updates, all engine images might need updates. Keeping a strong backward-compatible interface (e.g. a fixed REST API contract) mitigates this.
- *Client Integration:* Good, assuming the clients or an ops team are capable of containerizing the code. Clients have full freedom to use custom libraries inside their container. They deliver the container (e.g. push to Azure Container Registry) and inform the platform (perhaps via an onboarding API or simply by an agreed naming scheme). The platform can then launch it in the cluster. No need to touch core code. If using a naming convention (e.g. engine name "foo" corresponds to DNS `engine-foo.default.svc.cluster.local`), even the registration step could be minimal (just deploy with correct name, and the system will find it by constructing the URL from engine name). This approach does require DevOps savvy from clients (or additional support from the platform team to manage deployments). However, this aligns with many modern MLOps practices where models are delivered as container images.

Kubernetes-Native Jobs (No Celery):

- *Scalability:* Potentially excellent in throughput (lots of pods launched as needed). However, startup latency and cluster churn must be accounted for. This is elastically scalable but might need careful tuning to avoid flooding the scheduler. It's ideal for batch processing scenarios or spiky workloads. If using a system like Knative or KEDA, it can auto-scale to zero and back up on demand. For consistently high QPS real-time inference, this might underperform a long-running service (due to constant cold starts).

- *Security:* Excellent. Each task in its own container is the ultimate isolation (similar to FaaS – Function as a Service – security model). Multi-tenant security is enforced by the cluster. The same container hardening measures as approach 3 apply. One slight risk: if jobs run on shared nodes, one could try to exploit the node kernel or hardware. This risk exists in approach 3 as well, and is mitigated by Kubernetes and container security best practices (run as non-root, etc.). Running each client's jobs on separate node pools could be an extra isolation layer if needed.
- *Maintainability:* Complex. Using K8s jobs means dealing with lower-level primitives. Retries, failures, and timeouts must be handled via Kubernetes job specs and perhaps custom logic (Celery, by contrast, has convenient retry semantics in code). Debugging why a particular job failed might require checking Kubernetes events and logs of a short-lived pod, which is a different workflow than developers are used to with Celery. Also, if multiple engine types are run as jobs, you either need a common job runner or many separate job definitions. It could become an ops burden to maintain CRDs for each new engine (though something like Argo could help manage various workflows).
- *Client Integration:* If each engine is a separate image, clients still need to containerize it. But instead of a long-running service, their container is invoked per task. Clients might not even need to provide a server – they could just write a script that processes input from an environment variable or file, since the job container can be invoked with the data. The platform would handle passing the data in. This is somewhat simpler for the client (no need to implement a web server), but the packaging and interface are custom (e.g. writing a small loop in a Docker entrypoint to fetch one message). Alternatively, a **generic worker image** could be provided by the platform where clients just drop in a model file and the rest is handled. However, that starts resembling approach 3 (a shared runtime image).

In summary, **Approach 3 (Containerized External Services)** offers the best balance for this scenario. It scores high on security and reasonably on scalability and maintainability, while keeping integration steps manageable. **Approach 4 (K8s Jobs)** is highly secure and decoupled, but might introduce latency and complexity issues that are unnecessary if we can achieve similar results with persistent services. Approaches 1 and 2 (in-process) are largely ruled out due to security and isolation shortcomings, despite their simplicity.

Recommended Solution: Containerized Engine Services with Dynamic Routing

Recommendation: Implement a **plugin-like system at the infrastructure level** by treating each client's ML engine as a microservice (container) and dynamically routing tasks to the correct service. The core application will act as an orchestrator that delegates inference work to the appropriate engine service. Celery will be retained initially as the task queue mechanism (for familiarity and ease of implementation), but the Celery workers will be transformed into lightweight request forwarders rather than doing heavy ML computations themselves. In the future, the Celery layer could be phased out in favor of direct event-to-service invocation or Kubernetes job triggers once the containerized approach is proven (a possible evolution path).

Why this approach best meets the requirements:

- **No core changes per engine:** The core system will be implemented to route based on engine identifiers, so it's generic. Adding a new engine means deploying a new container; the core doesn't need code changes or redeploy. This satisfies the deployment independence requirement.

- **Scalability:** Each engine runs in its own Deployment that can be auto-scaled (using HPA or KEDA) based on its specific load (CPU, requests, or queue length if it uses its own queue). The Redis + Celery combo will continue to buffer tasks and scale the number of routing workers as needed, but the heavy lifting is in the engine containers. This decoupling means we scale out what is needed: if one model becomes popular, we scale its service only, without wasting resources on idle parts. This is similar to how model-serving platforms scale containers per model on demand [13] [14].
- **Security:** Engines are isolated by container boundaries and (optionally) by namespace. We will enforce strict policies: engine containers run as non-root, have no access to the host filesystem, and only expose a minimal API. They cannot directly touch the core application's resources except through that API. If needed, we can even run client engines on separate node pools (e.g. using taints/tolerations or separate AKS node pools per tenant) to ensure stronger isolation. Code execution happens in a sandboxed environment (the container) which is exactly the recommended way to run untrusted user code [2]. We will also discuss code signing and image scanning to ensure the container itself is safe.
- **Maintainability:** The architecture is modular. The core team maintains the core API, queue, and orchestrator logic. Clients maintain their own engines. Clear interfaces (HTTP/gRPC) define the interaction. If an engine needs debugging, it's isolated to that service's logs and metrics. Versioning can be handled via API versioning if needed. Also, since engines are separate, one can upgrade a library for one model without risking others.
- **Developer Experience (Client Integration):** Clients get freedom to use their preferred environment in their container. To integrate, they follow a guide (provided by the platform) to implement the standard interface (for example, an HTTP endpoint `/predict` that accepts a JSON payload with the input and returns the prediction result). They then containerize this (possibly starting from a provided template Dockerfile or using a base image that already has the interface scaffold). They deploy it (the platform could offer a CLI or portal to do this deployment into AKS). Once deployed and registered, the client can send tasks labeled with their engine name and see them executed. This process is fairly standard in MLOps: it mirrors how one would deploy models to a serving cluster (like using BentoML to create a model server container and deploying it, or using Azure ML endpoints – but here it's within our controlled AKS environment).

Potential Diagram: The new architecture is illustrated in **Figure 2**. The REST API enqueues tasks with a field indicating the engine (e.g. engine "X"). Celery workers (now just lightweight routers) consume tasks and then call out to the corresponding engine service (Engine X Deployment). The engine performs inference and returns the result, which the worker then stores or routes back to the client (through the API or a result queue). Each engine is isolated, and new engine deployments can be added independently.

Figure 2: Proposed architecture with containerized engines. The core application enqueues tasks to Redis with an engine identifier. Celery workers pop tasks and delegate processing to the specified engine's service (via an internal API call). Each engine runs in its own container (or set of pods) and returns results which are stored for retrieval. The core app remains unchanged when new engines are introduced; only a new engine deployment is added to the cluster.

Implementation Guide for the Selected Approach

This section provides a step-by-step guide to implementing the recommended solution, including code snippets, configuration (YAML), and instructions for client developers.

1. Architecture and Data Flow

When a request comes in to the REST API, it includes or is mapped to an `engine_id` (or name) indicating which ML model/engine to use. The flow will be:

1. **REST API Layer:** Validate the request and enqueue a task to Redis. The task payload contains the input data (or a reference to it) and the target `engine_id`. For example, a JSON task message might look like: `{"engine": "clientX_v1", "input": {...}}`.
2. **Celery Router Task:** We define a generic Celery task, say `route_inference(engine_id, input_data)`, which all such messages are dispatched to. This task doesn't do the ML computation itself but will dynamically route to the appropriate engine service:

```
import requests

@app.task(bind=True, max_retries=3)
def route_inference(self, engine_id, input_data):
    try:
        # Construct the engine service URL (assumes DNS naming convention)
        service_url = f"http://{engine_id}:8000/predict"
        response = requests.post(service_url, json={"data": input_data},
timeout=30)
        response.raise_for_status()
        result = response.json()

# Save result to DB or cache (for example, directly returning might store
in Celery result backend)
        return result
    except requests.RequestException as e:
        # Optionally implement retry logic for transient failures
        raise self.retry(exc=e, countdown=5)
```

   In this example, we assume each engine's service is reachable via an internal DNS name equal to the `engine_id` (e.g. engine `clientX_v1` corresponds to Kubernetes Service `clientX-v1` on port 8000). We also include basic retry logic in case the engine service is temporarily unavailable (container starting up or transient network glitch).
3. **Engine Service:** The engine container will run a small web server to handle `/predict` requests. For instance, if using FastAPI or Flask in Python, the client might have:

```
from fastapi import FastAPI
import joblib  # just as an example for loading a model

app = FastAPI()
model = joblib.load("model.joblib")  # load model artifact at startup

@app.post("/predict")
def predict(request: dict):
    data = request["data"]
    # ... perform inference
    result = model.predict(data)
    return {"result": result}
```

This server would be packaged in the client's container image. It should be configured to listen on a known port (8000 in our example) and ideally only accessible within the cluster (no external exposure).

4. **Result Handling:** When `route_inference` gets the result, it can either return it (if using Celery result backend for synchronous retrieve) or store it in a shared database/cache. In a typical web app, the initial API call might have returned a job ID to the client. The client can then call a results endpoint with that ID. The REST API can fetch the result from where the Celery task saved it (Redis, DB, etc.) and return to client. This part can remain similar to how results were handled before, just that now the task's work was done out-of-process.

The data flow ensures the core app is mostly orchestrating and not doing ML math – that's all in the engine services.

2. Deployment of Engine Containers (Kubernetes YAML/PowerShell)

Each engine provided by a client will be deployed to the AKS cluster. This can be done via Kubernetes manifests or automated pipelines. A sample Kubernetes Deployment and Service for a client engine might look like:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: clientx-v1-engine
  labels:
    app: clientx-v1-engine
spec:
  replicas: 1  # start with one, can be auto-scaled via HPA or manually
  selector:
    matchLabels:
      app: clientx-v1-engine
  template:
    metadata:
      labels:
        app: clientx-v1-engine
    spec:
      containers:
      - name: engine
        image: myregistry.azurecr.io/clientX-engine:v1.0  # Client's container
image
        ports:
        - containerPort: 8000
        # Security context to enforce non-root, limited permissions
        securityContext:
          runAsUser: 1000
          runAsGroup: 3000
          allowPrivilegeEscalation: false
          capabilities:
            drop: ["ALL"]
        resources:
          requests:
            memory: "500Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1"
        env:
        - name: ENV
          value: "prod"
        # (Mount model files or config maps if needed)
---
apiVersion: v1
kind: Service
metadata:
  name: clientx-v1  # This is the DNS name the Celery task will call
spec:
  selector:
    app: clientx-v1-engine
  ports:
  - port: 8000
    targetPort: 8000
    protocol: TCP
```

In this YAML: - We deploy the client's engine container from Azure Container Registry (`myregistry.azurecr.io`). The image contains the ML model and server. - We expose it internally with Service name `clientx-v1`. The Celery worker uses this name to send requests. - Security context is applied to run as non-root and drop privileges (part of best practices). - Resource limits are set to prevent one engine from using unlimited resources on a node.

If using Helm or another tool, this could be templatized. Also, if each client gets their own Kubernetes Namespace, we would deploy the engine in that namespace (and the core app's code would need to call the service in that namespace, e.g. `http://clientx-v1.clientX-namespace.svc.cluster.local`). A simpler approach is to keep them in the same namespace but differentiate by name.

Auto-scaling: We can attach a HorizontalPodAutoscaler to this deployment to scale based on CPU or request per second. Alternatively, use KEDA with a queue trigger if the engine pulls from a queue. For example, if each engine service also connected to a queue (less likely in our recommended design), KEDA could scale them by queue length.

Sidecar for metrics/logging: Each engine pod could have a sidecar for standard concerns (like exporting Prometheus metrics or pushing logs), but that's orthogonal to core functionality.

3. Client Engine Development and Registration

For clients to build and register their ML engines, we will provide clear guidelines:

- **Interface Specification:** Define what API the engine container must implement. The simplest is an HTTP POST endpoint (as shown above). We should specify input format (e.g. JSON with certain fields) and output format (JSON result). Alternatively, we could specify a gRPC service definition (proto file) if performance and type safety are desired.
- **Base Image or Template:** To make it easier, provide a base Docker image or an example project. For instance, a base image might have FastAPI installed and a scaffold that reads an environment variable for model path. However, many will prefer to build from scratch to include their specific libraries (e.g. TensorFlow, PyTorch).
- **Packaging Model Artifacts:** The client's Dockerfile should include their model file or code to download it. If models are large, one might mount a persistent volume or pull from cloud storage at container start. These details should be sorted out as part of onboarding.
- **Building & Pushing Image:** Clients build the image (e.g. via Azure DevOps or GitHub Actions CI pipeline) and push to a registry that the AKS cluster can access (Azure Container Registry or Docker Hub, etc.). They must follow the naming convention or otherwise inform the platform of the image and desired service name.
- **Deployment Registration:** We can simplify deployment for clients by having an **Engine Registration API** in the core app. For example, a client could call `POST /registerEngine` with a spec like image name, engine name, etc. The platform could then trigger the Kubernetes deployment (if we have permissions to do so from the app, say via an Azure Function or by the ops team manually). Alternatively, the client provides the Kubernetes YAML and the platform team applies it. The registration step also involves adding the engine to an internal registry so the REST API knows the engine is available (this might simply be an entry in a database or config map listing active engines).
- **No Core Code Changes:** The core routing logic is already generic. At most, adding a new engine might involve updating a configuration file mapping engine IDs to service URLs (if not derivable from name). But this can be designed to be dynamic: e.g. the Celery task could do a DNS lookup for `engine_id` – if the service doesn't exist, it will fail. So an engine is "registered" by the mere act of deploying its service. For user-friendliness, though, we'd likely maintain a list of available engine IDs to validate incoming requests (to quickly reject if an engine name is unknown). This list can be updated via the registration process.

Engine Versioning: In the engine name (like `clientX_v1`), we can encode versions. A client can deploy a new version alongside the old (with a different name) if backward compatibility is needed, and then switch requests to the new one when ready.

Testing: Clients should test their container thoroughly locally (we can provide a dummy input example). They can also deploy to a staging environment of the cluster if available. Since their engine is accessed via a well-defined API, it's easy to run end-to-end tests: send an example request to their service and see if it returns the expected output.

4. Modifications to Celery (if retained)

We are retaining Celery for now as the mechanism to buffer and distribute tasks. The modifications needed are:

- Define the single generic `route_inference` task (as shown in code earlier). This task can be part of the existing worker code. It will use the requests library (or httpx, etc.) to call external engines.
- Remove or refactor the existing specific engine calls. Previously, workers directly executed the "standard engine." Now, that code can either be repurposed into its own engine service (the standard engine becomes just another service that might be the default), or it can be inlined in `route_inference` as a fall-back for `engine_id == "standard"`.
- Configure Celery to accept tasks for multiple queues if we decide to segment by engine. However, a simpler approach is to keep one queue but just have the message specify the engine. The Celery workers are all identical and call out accordingly. This avoids needing separate Celery worker pools per engine, which would be like going back to static allocation.

One might consider leveraging Celery's routing or chords for parallelism, but that's not needed here. A straightforward `delay(engine_id, data)` enqueuing to the default queue is fine.

Performance considerations: Because the Celery worker will now potentially sit idle waiting for an HTTP response from the engine service, we might want to increase the number of Celery worker concurrency (since each worker isn't doing CPU work while waiting on I/O). Python's GIL won't block on I/O, but Celery uses processes by default. It may be beneficial to enable gevent or eventlet in Celery for such a proxy use-case to handle many simultaneous requests efficiently (essentially turning the worker into an async IO client). However, given each task likely corresponds to a substantial ML inference, it might be fine to just run one thread per process and let them block – the throughput can be managed by scaling workers if needed.

Error handling: If an engine service is down or returns an error (e.g., model crash), the Celery task can retry or mark the task failed. Retries with backoff are appropriate for transient issues. If a particular engine consistently errors out, we might move that to a dead-letter queue or alert the client.

5. (Optional) Removing Celery – Kubernetes Native Route

While we are initially keeping Celery, it's worth outlining how one could transition to a more cloud-native approach in the future:

- Use **KEDA ScaledObject** on the Redis queue to auto-scale a **deployment of lightweight router pods** (similar to Celery worker but perhaps simpler Python script or Go service) which does the same as `route_inference` (pull from Redis, call engine, push result). This removes Celery's overhead but keeps Redis. Essentially, it's reimplementing a minimal worker. This might not be necessary if Celery is working fine, as Celery already integrates with KEDA.
- Or use **KEDA ScaledJob**: define a ScaledJob that launches a Kubernetes Job for each message [15]. The Job's pod could directly run the engine container to process the single task (if each message includes engine info). Achieving that might require a custom entrypoint that knows how to fetch the message. Realistically, ScaledJob is better used per engine queue. For example, have KEDA watch `queue_engineX` and spawn jobs running the `clientX-engine` image for each message. This way, you don't need a persistent service for engine X; it will spin up on demand for each task and

then terminate. This is truly serverless per request. The trade-offs in latency and overhead have been discussed earlier.

- Use a higher-level workflow engine like **Argo Workflows**: The REST API could submit a workflow that has a step “run container X with this input.” Argo would manage the rest (executing and persisting output). This might be overkill unless we have multi-step workflows.
- **Azure Container Instances / Functions**: Since we’re on Azure, another option is to offload execution to Azure Functions or Container Instances triggered by a queue message. For example, an Azure Function could be written to handle a queue event and call a specific container. This moves compute outside of AKS on a per-call basis. However, integrating that with our architecture might complicate the deployment (two orchestrators, one inside AKS and one outside). It’s an option if we want to avoid running client containers in our long-running cluster altogether (for cost or isolation reasons), but it introduces network latencies and possibly higher cost per execution.

The current plan is to use the **“call external service” approach** via Celery because it’s simpler to implement with the existing system and test incrementally. Once containerized services are in place, one can evaluate if Celery is adding too much overhead or if job-per-request would be more efficient for certain engines (especially if they are rarely invoked – one could imagine not running some engine service at all until needed, which KEDA can handle by scaling to zero or using ScaledJob).

The flexibility of the design is that the engines are containerized and stateless. We can choose different orchestration without changing the engines themselves. That decoupling is key: we could swap out the Celery+Redis layer with another messaging or event system in the future, and the engine services wouldn’t need modification as long as they receive the same input.

Security Best Practices

With the recommended architecture, the primary security focus is on **sandboxing the execution of untrusted client code in their containers** and protecting the rest of the system. Here are best practices and measures:

- **Container Isolation:** Each engine runs in a container with a locked-down runtime. Use Kubernetes SecurityContext settings: run as non-root user, disallow privileged mode, drop Linux capabilities that are not needed (as shown in the YAML snippet, dropping ALL and no privilege escalation). This ensures even if the code tries something malicious, it has minimal OS permissions. The filesystem in the container should be read-only if possible (except perhaps a /tmp mount if needed), to prevent tampering with the environment at runtime.
- **Network Policies:** Implement Kubernetes NetworkPolicies to control traffic. For example, engine pods might only be allowed to send network requests to the core services (API/Redis) and perhaps external APIs if necessary for their function (if a model needs to call an external data source, that might be allowed on a case-by-case basis). They should not be able to talk to each other’s pods or to the database directly. In our case, ideally an engine only accepts connections from the Celery workers (or API) and maybe egress to the internet is blocked unless required. Limiting egress prevents a malicious engine from exfiltrating data or scanning the network.
- **Resource Quotas and Limits:** By assigning CPU/memory limits per engine, we prevent a DoS where one engine hogs all resources. Also consider using Kubernetes ResourceQuotas if clients are in separate namespaces, to cap total resource usage per client. Additionally, GPU access should be

controlled (if some engines need GPUs, give them specific GPU quotas and ensure others can't use GPU nodes).

- **Namespace Multi-Tenancy:** If security between clients needs to be absolutely strict, run each client's engines in a separate namespace and use Kubernetes RBAC to ensure they cannot access each other's resources. One could even run them in separate AKS node pools (with node taints) so that client workloads run on different underlying VMs. This provides isolation at the cost of resource fragmentation.
- **Image Security:** Require clients to follow secure build practices:
- They should use trusted base images (up-to-date with security patches).
- Encourage minimal images (so fewer tools that could be exploited are present).
- All images should be scanned (using Azure Security Center or similar) for vulnerabilities before deployment. The platform can enforce that only images passing certain checks (and perhaps signed by the client/developer) are allowed to run (using image admission controllers or Azure Policy for AKS).
- Optionally, use Docker Content Trust or sign images (e.g. with Notary or Cosign) so that the cluster only pulls verified images.
- **Engine Interface Hardening:** The engine services will expose an API – we should ensure it's not exposing more than needed. Ideally, the engine process should not accept arbitrary commands – just the one endpoint for inference. Input validation is crucial (the engine should validate the input format to avoid, say, code injection via the input). If using JSON, use proper parsing. If the engine uses a dynamic language, be cautious about `eval` on input, etc.
- **Audit and Monitoring:** All calls from core to engines should be logged (which engine was called, when, and how long it took, plus success/failure). Kubernetes audit logs should track if any unusual access is attempted by an engine container. Implement logging inside engine containers as well (clients should log important actions). A centralized logging system will allow detection of anomalies (e.g., an engine container suddenly making lots of outbound requests or throwing errors).
- **Timeouts and Circuit Breaking:** The orchestrator (Celery task or whatever calls the engine) should enforce timeouts. In our example, we set `timeout=30` seconds on the request. This prevents a stuck engine from hanging the worker indefinitely. If an engine times out consistently, the system could trip a circuit breaker (stop sending new tasks there and mark it unhealthy, notifying the ops team or client).
- **Data Security:** Ensure that one client's data is not accessible by another's engine. This mostly comes down to not sharing volumes or databases across tenants. The input to an engine should only contain that tenant's data (which is handled at the application level). If results are stored in a common DB, use row-level security or at least namespace the data by tenant.
- **Secret Management:** If engines need secrets (e.g., to access an external API or a license key), use Kubernetes secrets and mount/inject them only into that engine's pods (and restrict access via RBAC). Do not bake sensitive info into images.
- **Pentesting and Fuzzing:** Treat each engine service like an external application – consider fuzz-testing the interface to ensure it can't be easily crashed by malformed input. Also, periodically run security tests (even though they are internal services, a compromised core could try to attack an engine or vice versa – trust boundaries are somewhat blurred in internal network, so zero-trust principles can be applied).
- **Backup plan for malicious engines:** In case a client does deploy a malicious container (intentionally or through being compromised), have mechanisms to quickly isolate or shutdown that container. This could be manual (devops team kills the deployment) or automated (detect if an engine is making disallowed calls via network policy alerts or unusual resource patterns, then cut it off).

> Running engines with a seccomp profile that blocks syscalls related to filesystem and networking beyond what's needed is also a powerful mitigation.

By following these practices, the platform will significantly reduce the risk that running user-provided code entails. Essentially, we treat the engines as untrusted services in a zero-trust network inside our cluster.

It's worth noting that **this container isolation approach is widely used in industry** – for example, AWS Lambda runs user code in isolated containers/VMs, Azure Functions can run in isolated worker processes, and SaaS platforms often allow user extensions via webhooks or external services rather than running in-process. Our solution aligns with that philosophy by giving each client engine its own sandboxed execution environment.

Conclusion

In summary, to enable dynamic integration of client-provided ML inference engines on AKS, the best solution is to **decouple the engines from the core application by containerizing them and employing a dynamic routing/orchestration layer** to invoke the correct engine per task. This architecture (1) meets scalability demands by independently scaling each engine service, (2) upholds security through strong isolation of untrusted code [2] [16], (3) remains maintainable and extensible as new engines can be added without altering the core, and (4) provides a clear path for client integration with minimal friction (standard interfaces and deployment procedures).

Alternative approaches like in-process dynamic loading or Python plugins were considered but deemed high-risk due to Python's lack of sandboxing and the potential impact on stability [1]. The containerized approach leverages proven practices of microservices and ML model serving [7], essentially treating each model as a service. While it introduces some complexity in deployment, this is manageable with Kubernetes tooling and is outweighed by the gains in isolation and flexibility.

The current Celery/Redis infrastructure can be adapted to this model by turning workers into task routers that call engine services. In the future, the system could evolve towards a more Kubernetes-native event-driven execution (using KEDA or similar) for even finer-grained scaling [15], especially as the number of engines grows.

By implementing the guidelines and best practices detailed in this report – from code examples of how to dynamically call engines, to YAML templates for deploying engines, to strict security configurations – the organization can enable clients to plug in their custom ML models safely and efficiently. This empowers clients with independence and customization, without compromising the stability or security of the core platform. The result is a multi-tenant ML serving architecture that is robust, scalable, and agile in the face of evolving client needs.

---

[1] [2] [4] web applications - Safe Plugin Architecture for Python Web API - Software Engineering Stack Exchange
https://softwareengineering.stackexchange.com/questions/447610/safe-plugin-architecture-for-python-web-api

[3] [16] Security of Django application running untrusted code via Celery - Deployment - Django Forum
https://forum.djangoproject.com/t/security-of-django-application-running-untrusted-code-via-celery/22631

[5] [6] Architectural approaches for AI and ML in multitenant solutions - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/approaches/ai-ml

[7] [8] [13] [14] Best Tools For ML Model Serving
https://neptune.ai/blog/ml-model-serving-best-tools

[9] [15] Scaling Jobs | KEDA
https://keda.sh/docs/2.9/concepts/scaling-jobs/

[10] [11] Celery on k8s alternative : r/kubernetes
https://www.reddit.com/r/kubernetes/comments/145v2rs/celery_on_k8s_alternative/

[12] Scalable ML Inference Pipeline project, which involves using Knative, Ollama (Llama2), and DAPR on Kubernetes | by Simardeep Singh | Medium
https://medium.com/@simardeep.oberoi/scalable-ml-inference-pipeline-project-which-involves-using-knative-ollama-llama2-and-dapr-on-715b422919a1