What to check before adding a new AI model to your app

Adding a new model to an application is rarely just a matter of swapping one name for another. The model may accept different inputs, return different output shapes, cost more time per request, or fail under conditions the old one handled well. A careful review before launch saves awkward fixes later and helps the app behave in a predictable way.

The useful question is not only "Does this model work?" but "Does it fit the way the app already works?" That means checking the contract between the app and the service: what goes in, what comes back, what happens when something goes wrong, and what the user sees while waiting. These points matter whether the app handles text, images, audio, video, or a mix of them.

Define the job before touching the code

Start by writing down the exact task the new model is meant to perform. "Generate answers" is too broad. "Summarize a support ticket into three short bullets" gives the integration a clear target. A tight definition makes it easier to judge output quality, choose prompt templates, set limits, and decide which failures should be shown to the user.

It also prevents feature creep. A model that is strong at classification may be a poor fit for long-form writing, and a video model may need different handling from a text endpoint. If the app needs several kinds of output, treat each one as its own path with its own checks. Shared code can still sit underneath, but the acceptance criteria should remain specific.

Check access, credentials, and environment setup

Confirm that the API key is stored outside the codebase and loaded from an environment variable or a secret manager. Hardcoded keys create avoidable risk, especially when code is copied between local machines, staging, and production. Separate keys for separate environments also make it easier to trace usage and rotate credentials without breaking everything at once.



Check that the application uses the correct base URL and that the service expects authentication in the format the client sends. Some services use a bearer token, others use a header with a different name. A small mismatch can produce errors that look like model failures when the request never reached the model at all. Log enough detail to diagnose the request, but never log the key itself.

If the app depends on one provider for several media types, centralized access for AI tools can reduce the number of separate credentials and billing flows the team has to maintain. The practical benefit is simpler configuration, provided the application still handles each model's request and response format correctly.

Review the request schema field by field

Read the request schema carefully before building payloads. Pay attention to required fields, optional fields, allowed ranges, file formats, size limits, and any restrictions on prompt length or media duration. A field that seems minor, such as a response format flag, can change the shape of the result and break downstream code.

Pay close attention to how input is encoded. Text may need token limits, images may need a specific format or resolution, audio may need a supported sample rate, and video may need duration or file size checks before upload. Validate inputs in the app before sending them. That keeps invalid requests from consuming time and makes error messages more useful to the person using the app.

When several models sit behind one interface, avoid pretending every payload is the same. A shared adapter can normalize common features, but it should preserve differences that affect quality or cost. Clear internal naming helps here. A generic label like "generate" hides too much once the app supports several modes.

Plan for responses that vary

Model output is not always neat. A response may include metadata, usage details, safety flags, partial results, or an empty answer. The app should read the fields it needs and ignore the rest without assuming a fixed order or perfect formatting. If the output is shown to users, add a display layer that trims noise and handles missing content gracefully.

Decide what counts as a valid result. For a classifier, that may mean a label plus a confidence value. For a text model, it may mean a structured object that parses cleanly. For media generation, it may mean a file reference that can be fetched and previewed. Validation should happen before the result is stored, sent to another service, or presented as finished work.

Streaming responses need their own review. They can make an app feel more responsive, but they also introduce partial output, cancellation behavior, and reconnect edge cases. If the product does not need the immediacy, a standard response may be easier to support and test.



Set sensible limits and failure behavior

Every external model call can fail. Timeouts, invalid inputs, rate limit responses, service errors, and unexpected output all need a defined path. The app should distinguish between a user-correctable problem, such as an unsupported file, and a temporary service issue. The message shown to the user should explain what happened without exposing internal details.

Timeouts deserve particular care. A single slow call can hold up a request thread, leave a spinner running, or create duplicate jobs if users retry. Set a timeout that matches the task, add cancellation where the interface supports it, and decide whether retries are appropriate. Retries work best for transient errors and should avoid repeating requests that were rejected for bad input.

Check cost controls and usage visibility

Before release, confirm how usage is counted for the models the app calls. Text, image, audio, and video work can consume resources in different ways, and a change in prompt length, output length, resolution, or duration can alter consumption. Add app-level caps where possible, especially on public forms or automated jobs.

Usage visibility also helps with debugging. Record the model name, request type, status, latency, and relevant usage fields in a place the team can review. Avoid storing sensitive user content unless the product genuinely needs it. Good records make it possible to answer practical questions later, such as whether a spike came from one feature, one customer workflow, or a malformed request loop.

Test quality with real product inputs

Generic sample prompts only prove that a request can complete. Quality testing should use inputs that resemble actual app traffic, including awkward edge cases. For text features, check tone, formatting, accuracy, and instruction following. For media features, check whether the output fits the intended display area, file handling, and user expectations.

Keep a small set of repeatable cases that can be run whenever prompts, models, or parameters change. This does not need to be elaborate. A short collection of representative inputs with expected behavior can catch regressions that unit tests miss. Where output is subjective, use a simple review rubric so feedback stays consistent.

Also test what happens when the model is unavailable. A feature that degrades quietly may be better than one that blocks the whole page. Decide whether the app should show cached results, offer a simpler fallback, queue the job, or ask the user to try again. The right choice depends on how central the feature is to the workflow.



Verify deployment details before release

Configuration often differs between local development and production. Confirm that the deployed app has the needed keys, permissions, network access, and file storage. Check that temporary files are cleaned up, generated media is stored in the right place, and long-running jobs do not depend on a process that restarts too quickly.

Release in a controlled way. A small rollout, a feature flag, or an internal-only test gives the team room to watch logs and response quality before wider use. Keep a rollback path ready, especially if the new model changes stored data formats or user-facing behavior. A clean rollback plan is not pessimism. It is part of treating the model as a dependency that can change.

Finally, document the integration where future maintainers will find it. Note the model purpose, required fields, timeout rules, retry behavior, known limitations, and how to update credentials. Documentation should explain decisions, not repeat every line of code. When someone returns months later, the goal is for them to understand why the integration behaves as it does and what to check before changing it.