Utilizing OpenRouter Presets vs. Hardcoded Model “Cocktails”

Understanding OpenRouter Presets

OpenRouter **Presets** are named configurations that let you define all your Large Language Model (LLM) settings in one place, separate from your application code . A preset can encapsulate things like which model(s) to use (including an ordered list of models for failover), preferred providers, system prompts, and generation parameters . In essence, presets allow you to offload complex LLM configuration from your code to the OpenRouter dashboard. This provides several benefits: you can centrally define model selection and routing rules, update prompts or parameters on the fly without changing code, and focus your codebase on product logic instead of hardcoding model details . OpenRouter explicitly designed presets to support use cases like A/B testing different models or switching out models quickly – all by tweaking the preset in the web UI rather than deploying new code .

How to use Presets: Once you've created a preset via the OpenRouter dashboard, you can reference it in API calls in a few ways. The simplest is to treat the preset as a "virtual" model, using its slug. For example, if you have a preset slugged "my-cocktail", you can call the API with "model": "@preset/my-cocktail" in your request JSON . Under the hood, OpenRouter will apply all the configuration stored in that preset (including whichever underlying model or models it specifies, system prompt, etc.). You can also combine a preset with a specific model override (e.g. "model": "openai/gpt-4@preset/my-cocktail" to use GPT-4 but with other settings from the preset) or use a dedicated "preset": "my-cocktail" field alongside a base model . These options give flexibility, but in most cases you'd simply refer to the preset by its slug for cleaner code.

Hardcoded “Cocktails” in Code

By **hardcoded “cocktails”**, we mean manually implementing the logic of choosing models (and fallbacks) directly in your application code, rather than delegating that to an OpenRouter preset. For example, your code might have an if/else or configuration file that says: *ModelCombo A* uses GPT-4, if that fails then use Claude 2, otherwise use GPT-3.5; *ModelCombo B* uses a different chain, etc. You might capture user selection of a combo and then your code decides which underlying model(s) to call. This could even leverage OpenRouter's API directly with a models array parameter to handle fallback (since OpenRouter allows specifying an ordered list of models in a request without a preset) . In other words, it's technically possible to implement "cocktails" by hardcoding model lists and making the API call with that list each time, or by catching errors and retrying with alternate models in code logic.

Pros of the hardcoded approach: The main perceived advantage is control and transparency – you explicitly dictate in code which models are used and how failover happens. It might seem simpler if you only have one fixed strategy or a very custom routing that you manage internally. There's no external dependency on updating a preset; everything lives in your codebase.

Cons: Hardcoding model combos quickly becomes **fragile and inflexible**. Any time you want to adjust the combination (say swap in a new model version, reorder fallbacks, or tune a system prompt), you would need to change the code and redeploy. This tightly couples your model configuration to your release cycle. It's also more difficult to **A/B test or iterate** – trying a new model or parameter for just a portion of traffic would require adding conditional code. In short, embedding these "cocktail" definitions in code can lead to a maintenance headache, especially as you integrate more models or need to respond to model outages/updates. You'd essentially be re-implementing logic that OpenRouter presets are designed to handle for you (like routing and fallback), but with more manual work and risk of errors.

Using OpenRouter Presets for Model Combinations

*The OpenRouter preset creation interface allows you to configure a **cocktail** of models and settings. In this example, a preset is being created with a custom name and system prompt, and **Google's Gemini 2.5 Pro** selected as the default model. The interface allows adding multiple models; if more than one model is listed, they will be used as sequential fallbacks in the order given (as indicated by the UI hint). Once saved, this preset can be referenced by its slug in API calls.*

Leveraging **presets to create your "cocktails"** is the approach OpenRouter intended, and it directly addresses the requirements you outlined. You can create (via the OpenRouter web dashboard) a preset for each combination of LLMs you want to offer. For example, you might set up five presets representing your prepackaged LLM combos (e.g. *Combo Alpha*, *Combo Beta*, etc., each with a distinct mix of models). In each preset's settings, you can specify:

- **Primary model and fallback models:** In the preset editor, under "Model Selection," you can add multiple models. The first model in the list is the primary, and any additional models act as backups. OpenRouter will automatically route the request to the next model in the list if the primary is unavailable or returns an error . This failover happens transparently – if the primary model's provider is down, rate-limited, or even if the response is blocked due to content policies, the system will try the next model in your preset's list . You do *not* need to write code to catch the error and retry – the OpenRouter backend handles it. (If all models in the list fail, only then will you get an error back.) Importantly, the failover is **contained within the preset** as you desire – it won't jump to a completely different preset, but rather substitute the next model **within the same preset's list**. This aligns perfectly with your point (3) that *"the failover should sub models within the preset, not go straight to another preset."* In practice, you'll define the backup list inside the preset configuration, and OpenRouter ensures that sequence is followed for that preset's calls.

- **System prompts and parameters:** Each preset can also include a custom system prompt and default generation parameters (temperature, etc.) . This means your "cocktail" can encapsulate not just which models to use, but also *how* to use them (e.g. a specific role or persona via system prompt, or certain decoding settings optimal for that combo). For instance, if you have an **"UltrAI cocktail"** meant for a particular task or style, you could bake its special system instructions right into the preset. When users select that preset, every request automatically includes those instructions without extra code on your part.

- **Provider routing preferences:** If the same model is available from multiple providers (say OpenAI vs Azure, or different OpenRouter providers for the model), presets let you define preferences or sorting (by cost, latency, etc.) . This can add reliability and cost-optimization. However, if you're mostly using distinct models from different vendors in your combos, provider routing might be less relevant. Still, it's good to know the preset can manage provider selection too (for example, prefer your own hosted model first, then a cloud provider as backup).

Using presets for your combos means that your application code doesn't need to know the detailed model chain. You would simply take the user's choice of combo (e.g. via a dropdown or some selection UI) and map that to the corresponding preset slug. The API call then uses that preset – OpenRouter will handle sending the request to the appropriate model and falling back as configured. If at some point you realize you want to swap the order of fallback (maybe Model B should be tried before Model C), or upgrade to a newer model, you just update the preset in the dashboard. The **next API call will immediately use the updated settings**, since the preset is referenced by name and always resolves to its latest version . This gives a high degree of flexibility and **resilience**: your code remains unchanged while you can respond to model changes or outages in real time.

Pre-Packaged vs. User-Created Cocktails

It sounds like you have a roadmap for offering model cocktails to users in different ways, ranging from fixed choices now to user-customized combos in the future. Let's break down the scenarios:

1. **Pre-packaged Combos (Now):** In the current phase, you plan to present users with a set of (say) five predefined "cocktails" of LLMs. These could be combinations tuned for different priorities (e.g. one might prioritize speed/cost with a smaller model, another prioritizes quality with a larger model, etc.). For this scenario, using OpenRouter presets is ideal. You can create five presets, one for each cocktail, each locked to the models and settings you've chosen. Users then simply pick one, and your code calls the corresponding preset. The presets act as **"locked" configurations** – users cannot modify the internals (which addresses point (2) in your notes, since these are locked unless you eventually allow full custom combos). This approach ensures consistency: you know exactly which models and prompts are being used for each cocktail, and you can test and fine-tune them beforehand. It's also straightforward to implement in code (just a mapping of user selection to preset slug).

2. **UltraAI (Specific Default Combo):** You mentioned an "UltrAI cocktail" which could be a recommended or special combo in the future, perhaps auto-selected for certain tasks or as a default. This could simply be another preset (one that the system chooses by default, rather than the user actively selecting it). For example, you might have a preset that routes to what you consider the best general-purpose model (or even uses OpenRouter's openrouter/auto model router for dynamic selection). If it's task-specific, you might include a tailored system prompt. In practice, implementing this is again a matter of calling a preset – whether the user knows about it or it's under the hood, it's still the same mechanism. Presets make it easy to adjust the UltraAI combo over time (say, swap in a more advanced model when available) without the app needing an update.

3. **User-Created Combos (Future):** Ultimately allowing savvy users to **"create their own cocktail"** is a bit more involved, but also feasible. There are a couple of ways you could support this when the time comes:

    - **Via OpenRouter UI:** One option is to let advanced users log into an OpenRouter dashboard (or a connected interface) and create their own presets. However, this might not be ideal for a consumer-facing product, as it exposes a lot of complexity and requires the user to have some knowledge of models. It's more appropriate if your users are developers or if you plan to expose the preset management through your own UI.

    - **In-App Model Selection:** A more user-friendly approach is to build a custom interface in your app where the user can pick and choose models from a list (and maybe set some parameters), and then **use that selection directly in API calls**. You don't necessarily have to create a saved preset for each user combination; instead, you could take the chosen models and call OpenRouter with the models array parameter on the fly. For example, if a user picks Model X with fallback Model Y, your code can issue the request with "model": "X" and "models": ["Y"] (or "models": ["X","Y"] with the first being primary) to achieve the same effect as a preset . OpenRouter will attempt X, and fail over to Y if needed, just for that request. Essentially, you'd be programmatically doing what a preset would otherwise store. This gives users total freedom in constructing a combo, without needing you to predefine it. The downside is you'll need to impose some limits or validation (to avoid impossible combos or ensure the user has access to those models), and the onus is on your code to supply all necessary parameters (system prompt, etc., which you might default or also let them customize).

  - **Hybrid**: You could combine the above approaches by offering a set of base presets that users can clone or tweak. For example, start from a preset and then allow modifications. Currently, OpenRouter's API itself does not (to our knowledge) provide an endpoint to create/update presets programmatically (presets are managed through their dashboard UI) . So an *in-app* custom cocktail creator would likely use the direct models parameter method rather than trying to script preset creation. In the future, if OpenRouter introduces an API for managing presets, your app could leverage that to create user-specific presets behind the scenes. But until then, handling it in code is perfectly fine for the custom scenario.

In summary, **right now (as of 2025)** your best course is to use **presets for the fixed set of cocktails** you offer to all users. These are the "prepackaged LLM combos" you mentioned – they are curated by you, likely "locked" in terms of configuration. Users choose from these options but cannot alter their makeup. This gives a controlled experience and lets you ensure each preset is well-tested. Down the road, if you want to give users full freedom, you can extend your system to accept a custom model list (or potentially integrate with preset-creation if available). The preset approach you use now will not hinder that; in fact, it sets a good foundation by cleanly separating model config. You might even expose something in the UI like "Advanced: build your own combo" which then uses a different code path (bypassing the preset list and using dynamic models param). This way you maintain the **simplicity and reliability of presets for the majority**, while having a path for power users later.

Implementation for Resilience and Maintainability

To embed this in code **resiliently**, you should minimize hardcoded assumptions about models and instead lean on presets as configuration references. Here are some recommendations:

- **Use Preset Slugs in Config**: Treat the preset slug (or an identifier for each cocktail) as a configuration variable. For example, your application could have a mapping like {"combo_easy": "@preset/easy-combo", "combo_power": "@preset/power-combo", ...}. When a user picks a combo (or when you default to UltrAI), your code just looks up the corresponding preset slug and makes the API call. If you ever need to change what "combo_easy" means (e.g. swap models), you do so by editing the preset in OpenRouter, not by changing this mapping. As long as the slug stays the same, your code continues to call @preset/easy-combo and now magically it uses the new underlying model or prompt you configured. This decoupling makes your app more maintainable and able to **rapidly adapt** to new models or parameters .

- Failover Handling: Rely on OpenRouter's built-in failover within presets. You do not need to implement a secondary call in your code when a model fails – OpenRouter will automatically try the next model in the preset's list if the primary returns an error . This covers many failure modes (downtime, rate limits, even content filtering) without extra logic on your part . It's still prudent to handle an ultimate failure (if all fallbacks fail) gracefully – e.g., return an apology to the user or try a very basic model as a last resort if appropriate. But these should be generic error handlers; the *specific* fallback sequencing is handled by the preset. By keeping failover logic in one place (the preset config), you reduce complexity and potential points of failure in your code.

- Hybrid Code Paths: If you plan to add the custom cocktail functionality later, design your code to have an abstraction for "model selection". For instance, have a function that given a user's choice returns the appropriate API call parameters. If the choice is one of your predefined combos, it returns model: @preset/X. If the choice is a custom list (in future), it returns models: [A, B, C] plus perhaps a base model. This way, the high-level flow (sending a chat completion request) doesn't care if the source was a preset or an on-the-fly combo – it just gets a request payload. This abstraction will make your system resilient to change because you can extend or modify the selection logic without touching the core chat handling. In the near term, this function may be trivial (just mapping preset slugs), but having it will make a hybrid approach easier to implement later.

- Testing and Versioning: OpenRouter presets have version history and easy rollback options . This means you can tweak a preset's config (say adjust a prompt or swap a model) and if something goes wrong (e.g., the new model behaves poorly), you can revert to a previous version of the preset. From the API perspective, the preset slug always points to the latest version , so you don't need to change anything client-side to roll forward or back. Embrace this by testing changes in non-production presets and then updating the live preset when confident. This feature contributes to resilience since you can respond quickly to issues (like a model misbehaving) by altering the preset without any downtime or code change.

- Avoiding Hard Dependencies: One consideration for long-term resilience is to avoid tying your logic too specifically to any one model or provider. Presets help here because you could, for example, include a mix of providers (OpenAI, Anthropic, etc.) as fallbacks. If OpenAI has an outage, your preset could automatically flow to Anthropic's model. If you had hardcoded direct calls to OpenAI in your code, implementing such provider switch-over is non-trivial and would likely be overlooked until an outage hits. By using presets, you've essentially outsourced that complexity – as long as your preset is configured with alternatives, the app stays operational even if one service fails. This kind of robustness is hard to achieve with purely hardcoded logic.

In conclusion, I strongly recommend utilizing OpenRouter Presets for managing your LLM "cocktails" rather than hardcoding them. Presets provide a clean separation of concerns and make your system easier to maintain and evolve . In the immediate term, define your five prepackaged combos as presets and let users choose among them. Your code then remains simple – it just references whichever preset is needed, and OpenRouter handles the rest (model selection, parameter application, and intra-preset failovers). This will give you a stable and resilient foundation: you can refine model combos or swap out models with minimal effort, and the built-in failover increases reliability for your users .

Looking ahead, this approach won't hinder future enhancements like user-created combos; you can always extend your system to accept dynamic model lists for power users when you're ready. That would be a complementary feature built on top of the same OpenRouter capabilities. In other words, using presets now is not an either/or choice but rather a best practice that you can augment with custom logic

later. By possibly combining both (presets for standard options, and a code-driven path for custom selections), you get the best of both worlds – ease of management for known good configurations, and flexibility for advanced scenarios. This hybrid strategy ensures your application remains **robust to changes** (new models, model outages, evolving user needs) and saves you from constant code churn whenever your LLM strategy updates.

Sources:

- OpenRouter Documentation – *Presets: Manage your LLM configurations*
- OpenRouter Documentation – *Model Routing and Fallback (models parameter)*
- OpenRouter Announcement – *Introducing Presets (Why Use Presets)* (separation of config and no-code updates)