ChatGPT

Modular File Design for AI Projects: YAML, Hydra, AIConfig and Best Practices

Introduction

When building AI applications — whether machine learning pipelines or LLM-powered agents — the choice of how to represent configuration, prompts and documentation has profound implications for maintainability and collaboration. YAML is widely used because it is human-readable and supports complex data structures, but it can become unwieldy when it grows large. JSON is machine-friendly but lacks comments and can be verbose. Markdown (`.md`) is excellent for human-readable documentation but cannot represent structured configuration without careful parsing. This report surveys modern best practices for breaking complex AI projects into modular files, drawing on recommendations from software-engineering frameworks such as Hydra and Kedro, DevOps guidance from GitLab and Spacelift, and emerging configuration standards for generative AI like AIConfig. The goal is to recommend a file and folder structure that balances modularity, clarity and machine-readability while being accessible to both AI frameworks and human developers.

Why use configuration files?

Complex AI systems often require parameters, model hyper-parameters, credentials and other settings that should not be hard-coded. Configuration files externalize these settings and thus:

- Separate code from configuration, allowing non-developers to modify parameters without touching Python/Java code [1].
- Make experiments reproducible; a saved config can be rerun to recreate the same experiment [2].
- Facilitate environment-specific settings (development vs. production) and parameter sweeps [3].

YAML has become the de-facto format for these files because it is concise and human-readable [4]. However, large YAML files can be difficult to debug, and over-nesting can obscure meaning. To mitigate these issues, best practice is to split configuration into smaller files and use tools/frameworks that support composition and overrides [5] [6].

YAML vs. Markdown vs. JSON

Format	Strengths	Weaknesses
YAML	Human-readable, supports comments and complex structures, widely supported in ML frameworks. Good for configuration and declarative representations.	Sensitive to whitespace; large files become hard to debug. Doesn't validate types; may interpret values like `NO` as boolean `False` unless quoted [7].

Format	Strengths	Weaknesses
JSON	Strict syntax, machine-friendly, widely used for APIs and training-data interchange. Tools can validate JSON schemas.	No comments; less human-friendly; verbose.
Markdown (.md)	Ideal for documentation, instructions and narrative. Supports headings, lists and links. Easy to read and write.	Not structured; cannot be parsed directly as config. Requires custom parsing if used to embed configuration blocks.

Recommendation: use YAML (or JSON) for configuration that must be read by code, and Markdown for human-readable documentation. When using YAML, keep files small and logically separated; avoid mixing machine-parseable config with long narrative content. If prompts or model specifications need to be versioned alongside configuration, frameworks like AIConfig store them in a JSON-serializable format [8] and should live in their own files.

Splitting configuration into multiple YAML files

Large monolithic configuration files are error-prone. Several authoritative sources recommend breaking them down:

- **Spacelift (YAML tutorial):** warns that although you *can* configure everything in a single YAML file, it becomes hard to debug; it advises splitting configurations into multiple files to speed up debugging [5] and to use descriptive keys and comments [9].
- **GitLab CI/CD guide:** encourages reuse of configuration through the `include` keyword. This mechanism includes external YAML files, enabling common variables or templates to live in a shared file and be included in project-specific YAML; it boosts readability of long files [6].
- **Hydra framework:** Hydra's key feature is dynamically creating a hierarchical configuration by composing multiple YAML sources and overriding them via command-line arguments [10]. Hydra encourages "config groups" where each component (e.g., database) has its own YAML file. A top-level `config.yaml` uses a `defaults` directive to specify which config groups to include [11].
- **Hydra's dynamic composition:** Hydra supports parameter sweeps and environment-specific configurations; you can override or add new values at runtime without editing the files [3].

Implementation pattern for splitting YAML

1. **Base configuration** – a `base.yaml` (or `default.yaml`) defines common settings such as default models, training hyper-parameters and file paths.
2. **Environment-specific files** – separate YAML files for `development`, `test` and `production` override only the keys that differ (e.g., database host, logging level). Tools like Hydra can merge these at runtime and handle overrides in the correct order [12].
3. **Component-specific files** – if your application includes distinct modules (data ingestion, model training, evaluation), create subdirectories (e.g., `model/`, `dataset/`) with YAML files for each component. Hydra config groups or GitLab `include` statements can assemble them.
4. **Experiment configurations** – place experiment-specific YAML files in a dedicated folder (e.g., `conf/experiments/exp01.yaml`). Each file only contains parameters you intend to change

between runs; they override the base config. Julien Beaulieu's project structure stores a default config and per-experiment configs that override the default using a merge function [13].
5. **Secrets and credentials** – keep credentials in environment variables or separate secret files; never in version-controlled YAML. Hydra's merge order can load `.env` variables with highest priority [14].

By following this pattern, you maintain separation of concerns, reuse common configuration, and avoid duplicate copy-paste across multiple YAML files.

Hydra: hierarchical config management

Hydra is an open-source framework from Facebook Research designed to manage complex configuration. Its documentation emphasises that Hydra can "dynamically create a hierarchical configuration by composition and override it through config files and the command line" [10]. This means you can:

- **Compose configuration from multiple sources.** A top-level config lists entries under a `defaults` section specifying which config files to use (e.g., `- db: mysql`). Hydra then composes these files into a single configuration object at runtime [11].
- **Override parameters from the command line.** Users can supply overrides like `python my_app.py db.user=root` to change a single field without editing the file [15].
- **Support multi-run sweeps** for hyper-parameter tuning. The `--multirun` flag will run the program multiple times with different parameter values and store results separately [16].
- **Group configurations into directories** (config groups) so that alternate implementations of a component (e.g., `db/mysql.yaml` and `db/postgresql.yaml`) can be swapped via the `defaults` directive [17].

Hydra is thus well-suited for projects with many configurable components, and it encourages splitting configuration into logically grouped YAML files rather than a single monolithic file.

YAML best practices

Authors of YAML tutorials and DevOps guides emphasise several practices that improve readability and maintainability:

1. **Consistent indentation** – YAML is whitespace-sensitive. Use spaces instead of tabs and be consistent (two or four spaces throughout) [18]. Editors and linters can automate formatting [19].
2. **Keep files short and modular** – large YAML files are hard to debug. Split configuration into multiple files and include them from a top-level YAML or via a framework like Hydra [20] [6].
3. **Meaningful keys** – choose descriptive key names so other developers understand the purpose of each entry [21].
4. **Comments and documentation** – annotate configuration sections with comments (prefixed with `#`) to explain purpose and constraints [22].
5. **Validation and linters** – use YAML linters or validators in your CI/CD pipeline to catch syntax errors early [23].
6. **Anchors and aliases** – YAML supports anchors (`&`) and aliases (`*`) to avoid duplication. However, anchors don't work across multiple files when using GitLab's `include` feature [24]. Use with caution.

Following these practices ensures that configuration remains clear and maintainable even as projects grow.

Managing generative-AI prompts and models

Traditional machine-learning configuration focuses on hyper-parameters and data paths. Generative-AI systems add the complexity of prompts, model parameters (temperature, system instructions, etc.) and chain structures. The **AIConfig** open-source project addresses this by storing prompts, model names, model parameters and metadata in a JSON-serializable file [8]. AIConfig aims to:

- Separate prompts and model behaviour from application code to reduce complexity [25].
- Store generative-AI artifacts in version control, enabling reproducibility and collaborative iteration [26].
- Support prompt chaining and multi-modal pipelines (e.g., connecting Whisper to GPT-4) by representing each prompt as a configurable node [27].

For LLM projects, storing prompts and model selection in a dedicated config file (AIConfig, YAML or JSON) and versioning it separately from code ensures that you can iterate on prompts without redeploying the entire application. Combining AIConfig files with YAML configuration (e.g., specifying which AIConfig file to load) offers a clean separation between high-level workflow and LLM specifics.

Kedro: a framework for reproducible pipelines

Kedro is an open-source Python framework designed to create reproducible, maintainable and modular data-science code. According to Neptune.ai's guide, Kedro accelerates pipelining and enhances reproducibility [28]. It applies software-engineering concepts such as reproducibility, modularity, maintainability, versioning and documentation [29]. The default Kedro project structure includes a `/conf` directory with configuration files for data catalog, logging and model parameters; a `/data` directory structured into layers; `/docs` for documentation; `/logs` for run logs, and pipeline code under `src` [30]. Kedro encourages splitting parameters into separate YAML files and uses a data catalog to define datasets. Its modular pipeline nodes are defined in separate Python files and referenced in pipeline configuration, promoting separation of concerns.

Recommended file structure for an AI project

Drawing on these sources, a robust and modular AI project repository might look like this:

```
project_root/
├── README.md              # High-level project overview and instructions
├── docs/                  # Markdown documentation for phases, design,
processes
│   ├── ideation.md        # Capture ideas and research notes
│   ├── poc.md             # PoC architecture, roles and design decisions
│   └── mvp.md             # MVP audit and transformation plan
├── conf/                  # Configuration files (YAML/JSON)
│   ├── base.yaml          # Default settings shared by all environments
```

```
│   ├── env/
│   │   ├── dev.yaml       # Development overrides
│   │   ├── test.yaml      # Testing overrides
│   │   └── prod.yaml      # Production overrides
│   ├── components/
│   │   ├── data_ingestion.yaml
│   │   ├── model_training.yaml
│   │   └── evaluation.yaml
│   └── experiments/
│       ├── exp01.yaml     # Experiment-specific overrides
│       └── exp02.yaml
├── prompts/               # Generative-AI prompts or AIConfig JSON files
│   ├── summarization.aiconfig.json
│   └── classification.aiconfig.json
├── src/                   # Source code for pipelines, agents, utilities
│   ├── data_ingestion/
│   ├── model_training/
│   └── evaluation/
├── scripts/               # CLI scripts or runners (e.g., Hydra / Kedro entry
points)
└── logs/                  # Run logs
```

- `README.md` **and** `/docs/` are in Markdown for human readers and capture ideation, PoC design, MVP planning and other narrative content. These files describe the "Watcher" role, guidelines, user requirements and design decisions, but they avoid embedding machine-readable configuration directly.

- `conf/` holds YAML or JSON configuration. A `base.yaml` defines defaults. Sub-folders separate environment overrides (`env`), component-specific files (`components`) and per-experiment configs (`experiments`). Hydra can compose these; GitLab's `include` can import them into CI pipelines; AI scripts can load them directly. Splitting the YAML this way keeps files short and logically grouped [5].

- `prompts/` contains AIConfig files (JSON) or YAML/Markdown prompt descriptions if you prefer to store prompts separately. Storing prompts in dedicated files allows prompt engineers to iterate without modifying the code base [31].

- `src/` **and** `scripts/` contain the Python code implementing your agents, data pipelines and CLI entry points. Each module references configuration via Hydra or another loader; parameters are never hard-coded. Kedro uses a similar separation with `conf`, `data`, `docs`, and `src` [30].

This structure aligns with best practices in DevOps and data science: it is modular, self-documenting, environment-aware and easy to extend.

Practical considerations for AI development

1. **Choose a configuration framework.** If your project will need hierarchical configuration, experiment sweeps and environment-specific overrides, adopt Hydra. It handles composition and overrides elegantly [10]. For simpler projects, plain YAML with `include` directives may suffice [6].
2. **Version control the configs.** Keep YAML/JSON/AIConfig files under version control. Each change to prompts or model parameters becomes traceable, which is crucial for reproducibility and governance [26].
3. **Separate secrets.** Do not commit API keys or credentials. Hydra allows `.env` files to override configuration at runtime [14]. In CI/CD, use secret management services.
4. **Document the "ritual."** Create a `ritual.md` or `CONTRIBUTING.md` that explains how to use the configuration, how to add new components, and how to run the pipelines. This ensures new contributors understand the multi-phase process from ideation to MVP.
5. **Use validation and testing.** Write unit tests for functions that load configuration. Use YAML linters and schema validation to catch errors before runtime [23]. In Hydra, you can register config schemas to validate fields.
6. **Iterate and refactor.** As the project grows, periodically audit the `conf` folder. Consolidate repeated patterns into base files; remove obsolete experiment configs; ensure naming conventions remain clear.

Conclusion

A robust AI-development workflow benefits from clear separation of configuration, prompts and documentation. YAML remains a powerful format for configuration, but unrestrained growth in file size leads to poor maintainability. Splitting configuration into modular files and using frameworks like Hydra or Kedro allows dynamic composition, environment overrides and reproducibility. For generative-AI projects, storing prompts and model settings in AIConfig JSON files decouples prompt engineering from application code [8]. Complement these machine-readable formats with Markdown documentation that captures ideation, PoC design and MVP planning. Following best practices in indentation, naming, commenting and validation [32], and leveraging features such as GitLab's `include` keyword and Hydra's config groups [6] [11], you can build AI projects that are maintainable, extensible and accessible to both developers and LLM-based agents.

---

[1] [3] Configs in Yaml and Hydra. In a previous story, I wrote about how... | by Hitoruna | Medium
https://medium.com/@hitorunajp/configs-in-yaml-and-hydra-a71dc116be8a

[2] [12] [13] [14] Building A Flexible Configuration System For Deep Learning Models · Julien Beaulieu
https://julienbeaulieu.github.io/2020/03/16/building-a-flexible-configuration-system-for-deep-learning-models/

[4] YAML Tutorial: Everything You Need to Get Started in Minutes
https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started

[5] [7] [9] [18] [20] [21] [22] [23] [32] YAML Tutorial : A Complete Language Guide with Examples
https://spacelift.io/blog/yaml

[6] [19] [24] 3 YAML tips for better pipelines
https://about.gitlab.com/blog/three-yaml-tips-better-pipelines/

8 25 26 27 31 AIConfig
https://aiconfig.lastmileai.dev/docs/basics/

10 11 15 16 17 Getting started | Hydra
https://hydra.cc/docs/1.1/intro/

28 29 30 Building and Managing Data Science Pipelines with Kedro
https://neptune.ai/blog/data-science-pipelines-with-kedro