Genealogy of ELIZA

I want to talk about Mr. Bayes and phylogentic trees,

Great — happy to! Tell me where you want to begin:

- **MrBayes itself** (the software: commands, blocks, priors, MCMC, diagnostics)
- **The underlying Bayesian phylogenetics** (tree priors, substitution models, posterior sampling)
- **Historical context** (Ronquist & Huelsenbeck's work, influence on modern Bayesian phylogenetics)
- **Practical issues** (convergence problems, mixing, partitioning, calibration, long-branch attraction)
- **Interpretation of output** (consensus trees, posterior probabilities, branch lengths, model selection)

Or if you prefer, you can describe what you're trying to do (e.g., "I have a FASTA alignment and want to infer a tree with MrBayes," or "I want to understand why Bayesian methods differ from maximum likelihood").

The underlying Bayesian phylogenetics

Great — let's build a clean, conceptual foundation for **Bayesian phylogenetics**, in a way that connects the core ideas behind MrBayes to the deeper statistical logic.

Below is the structure I recommend, and then we can zoom into any part in more detail.

---

1. The Core Bayesian Equation for Trees

Bayesian phylogenetics is built around:

$$P(\text{Tree, Model Parameters} \mid \text{Data}) \ \propto \ P(\text{Data} \mid \text{Tree, Model Parameters}) \times P(\text{Tree, Model Pa}$$

Which means:

- **Posterior** = what you want: the distribution of plausible trees.
- **Likelihood** = how well a given tree predicts the observed sequences.
- **Prior** = what kinds of trees and models you think are plausible before seeing data.

The key difference from ML/maximum parsimony:
Bayesian phylogenetics produces a *distribution* of trees, not a single best tree.

2. The Space of Trees Is Enormous

For n taxa, the number of unrooted binary trees is:

$$(2n-5)!!$$

This grows faster than exponential.
Even for 10 taxa → **2 million trees**.
For 50 taxa → **~10^{76} trees** (far more than atoms in the universe).

Thus:
Direct enumeration is impossible, so Bayesian methods use sampling via MCMC.

---

3. Likelihood: How a Tree Predicts Data

Given a tree with branch lengths + model, the probability of the aligned sequences is computed by:

- assuming a **continuous-time Markov process** of substitutions along each branch (Jukes–Cantor, HKY, GTR, codon models, etc.)
- using the **Felsenstein pruning algorithm** (dynamic programming up a tree, reduces computation from exponential to O(nk^2)).

The likelihood is the engine:
how probable are the sequences if this tree were true?

---

4. Priors (Often Under-discussed but Crucial)

Bayesian phylogenetics requires priors over:

Tree topology

Often a uniform prior across all labeled histories.
But note: uniform over *topologies* is *not* uniform over clade sizes.
(Many users don't know this; tree priors bias toward certain shapes.)

Branch lengths

Often exponential or gamma.
These heavily influence tree shape and the ability to detect divergence.

Substitution model parameters

E.g., base frequencies, rate matrices, gamma rate variation, etc.

Clock models (if used)

Strict, relaxed lognormal, relaxed exponential.

Priors play a real role when data are weak.

---

5. The Posterior Is Sampled by MCMC

MrBayes uses Metropolis–Hastings MCMC to sample trees and parameters.

Typical moves include:

- **NNI (nearest-neighbor interchange)**
- **SPR (subtree pruning and regrafting)**
- **TBR (rare)**
- **Branch length rescaling**
- **Model parameter updates**

MCMC generates a **chain** that moves through tree space, ideally proportional to the posterior.

The stored samples ≈ your posterior distribution.

---

6. Burn-in, Convergence, Mixing

Crucial concepts:

Burn-in

Early iterations are not representative; discard them.

Convergence

Is the chain sampling from a stationary posterior distribution?
Indicators:

- PSRF (potential scale reduction factor)
- trace plots
- ESS (effective sample size)

Mixing

How well the chain moves through tree space.
Bad mixing → huge uncertainty or stuck in local modes.

MrBayes helps with:

- **heated chains** (Metropolis-coupled MCMC)
- temperature tuning

---

7. Summaries: the Posterior Consensus Tree

From the trees sampled after burn-in, you build:

Majority-rule consensus tree

Each clade is included if it appears in ≥ 50% of sampled trees.

Posterior probability of each clade

Directly from the count of sampled trees containing that clade.

This is NOT the same as bootstrap values from ML.
Posterior probabilities are usually higher and have a different interpretation.

---

8. Why Bayesian Phylogenetics Is Popular

- **Uncertainty is quantified directly** (distribution over trees)
- Flexible priors for divergence times, tree shapes, clock models
- Integrates complex models more naturally than ML
- Better handling of ambiguous/missing data
- Posterior probabilities are intuitive and interpretable

---

9. Known Pitfalls

- Priors influence results more than users realize
- Mixing problems on large datasets
- Long-branch attraction can still occur under some models
- Posterior probabilities are often overconfident for short alignments
- Marginal likelihood estimation (for Bayes factors) is hard

Where do you want to go next?

- Derive the likelihood mathematically?
- Explore the MCMC proposals in MrBayes?
- Discuss priors for tree shapes (e.g., Yule vs uniform)?
- Compare MrBayes with BEAST, RevBayes, PhyloBayes?
- Talk about problems like long-branch attraction or heterotachy?

Just point the way!

Okay, so I want to think about doing this for computer programs, based on their source code.

Nice, this is a fun idea. Let's port the *Bayesian phylogenetics mindset* over to source code.

I'll sketch a concrete way to do "MrBayes for programs":

---

1. First: what plays the role of what?

In the phylogeny analogy:

- **Taxa** → individual *programs* (or versions, or repos, or functions/classes).
- **Characters / sites** → features of the source code.
- **States at sites** → values of those features (e.g., specific tokens, AST node types, API calls).
- **Tree** → a *genealogy of code*: who was forked from whom, who copied whom, etc.
- **Evolutionary model** → a model of how code changes over time (edits, refactors, renames, copy–paste).

You want:

$$P(\text{Tree}, \text{Model Params} \mid \text{Programs}) \propto P(\text{Programs} \mid \text{Tree}, \text{Model Params}) \times P(\text{Tree}, \text{Model Pa}$$

Exactly as with sequences, but now the "data" are source code artifacts.

---

2. The big headache: "alignment" for code

In biology you start from a multiple sequence alignment: positions 1..L are (mostly) homologous.

For code, there is no obvious 1D alignment:

- Code is **structured** (AST/CFG) not just a string.
- There is aggressive **recombination**: copy-paste, merging from multiple parents, library use.
- Refactors reorder functions, split/merge files, etc.

So before we get to Bayesian trees, we need a *representation of code as characters* that is at least *approximately* homologous across your taxa.

A few options

1. **Token-level pseudo-alignment**
    - Tokenize code (e.g., normalized identifiers, keywords, punctuation).
    - Maybe sort functions by name or some canonical ordering, then concatenate.
    - You'd get something sequence-like, but homology is weak and refactors will scramble it.
2. **Bag-of-features as "unordered characters"**

   - Define features like:
     - frequency of each opcode/API call
     - counts of certain AST patterns
     - cyclomatic complexity bins for functions
   - Discretize each into categories (0, 1–2, 3–5, >5, etc.).
   - Treat each feature as an independent character evolving over the tree.

   This is **uglier biologically**, but plays nicely with a standard “morphological” character model.
3. **AST-based alignment**
   - Parse all programs to ASTs.
   - Define a scheme to align nodes: e.g., corresponding functions by name, classes by name, or by structural similarity.
   - At each aligned AST node, define a character: node type, presence/absence of certain patterns, etc.

   This gives you fewer but more meaningful characters.
4. **Graph or embedding approach (conceptual for now)**
   - Embed each function or file (e.g., CodeBERT-like embeddings, or handcrafted graph features).
   - Discretize these into bins or clusters and use those cluster IDs as characters.

   This is more “modern ML”, but you’d still need to turn it into discrete characters for a MrBayes-like workflow.

---

3. A generative “code evolution” model

Once you have characters, you need an evolutionary model: a Markov process that changes character states along branches.

For code, a plausible model would include:

- **Point “mutations”**:
  - change an operator (`+` → `-`)
  - change a literal
  - change an API call
- **Renaming**:
  - systematic variable/class renaming (lexical but functionally neutral)
- **Indels**:
  - add/remove lines
  - add/remove functions or modules
- **Refactors**:
  - split function

  - inline function
  - move method between classes

For a first pass, you probably **don't** want to represent all of these explicitly. Instead:

- Define a **small state space of feature values** (like a morphological model).
- Use a continuous-time Markov process on those discrete states, just like Mk models in phylo.

Example: say one character is "# of calls to `foo()` per KLoC," discretized into 0,1,2,3+. Then your rate matrix Q governs transitions among {0,1,2,3+} along branches.

Later, you could get fancy and make the model closer to edit operations on ASTs.

4. Likelihood: P(program features | tree, parameters)

Once representation + model are fixed, the likelihood looks just like in normal phylogenetics:

- Characters (features) are assumed independent given the tree and parameters.
- For each character:
  - states at the tree tips = observed feature values in each program.
  - internal node states = unobserved.
  - probability of tip states given tree + Q computed with Felsenstein-like pruning, treating the character as a discrete trait.

So, mathematically, **nothing stops you** from running a standard discrete-character Bayesian phylo engine (like MrBayes) if you can:

1. Encode each program as a row of discrete characters.
2. Specify a reasonable Q (or family of Qs) for character evolution.

The "Bayesian" part is then entirely standard:

- Priors on tree topologies.
- Priors on branch lengths and Q parameters.
- MCMC over tree space, sampling (tree, Q, branch lengths) from the posterior.

5. Priors for software "trees"

What priors make sense?

- **Topology prior**
  - If you have no history: maybe uniform over labeled trees, as in biology.
  - If you know some things (e.g., Git history, project forks), that can inform topology priors or constraints.

  - You might encode a bias toward *star-like* radiations (a popular library being forked many times).
- **Branch length prior**
  - Could reflect *expected amount of change*: e.g., codebase size × time, or commit counts along the branch (if known).
- **Rate priors**
  - Priors that say most features are stable, but some are “hot”.
  - Could allow rate heterogeneity across characters (a gamma or lognormal mixture).

---

6. What is the *posterior* telling you for programs?

If you run such a model, your posterior over trees would answer:

- **Lineage / genealogy**: which program is descended from which; who forked whom.
- **Clades of related code**: clusters of codebases that share a lot of evolutionary history (e.g., malware families, near-duplicate repos, plagiarized assignments).
- **Uncertainty**: multiple plausible ancestry scenarios if the evidence is weak.

You can also:

- Map **character changes** to branches → “this is where they swapped HTTP library X for Y,” etc.
- Do **ancestral reconstruction** → infer earlier versions of code features (even if not in the repo history).

---

7. A concrete pipeline you could actually implement

Here’s a pragmatic, not-too-insane version 0.1:

1. **Choose your taxa**
   - e.g., 100 GitHub repos of implementations of the same algorithm.
   - or 200 student solutions to the same assignment.
   - or 150 malware samples suspected to be related.
2. **Extract features**
   - Parse code.
   - For each program, compute features such as:
     - counts of specific library/API calls (e.g., `socket`, `requests.get`, `println`, etc.)
     - presence/absence of particular idioms (e.g., list comprehension vs. for-loop).
     - distribution of function lengths (binned).
     - AST pattern counts (e.g., `if–else` depth patterns).
   - Discretize each into a small set of bins/categories.

3. **Build a character matrix**
    - Rows = programs.
    - Columns = features.
    - Entries = category IDs (like 0, 1, 2,...).
    - This is your “alignment” in MrBayes-speak.
4. **Run a standard discrete-character Bayesian phylogeny**
    - Encode this as a NEXUS data block with `datatype=standard` or `datatype=restriction` / `morph`.
    - Use MrBayes to:
        - estimate tree topology and branch lengths
        - estimate feature evolution rates (Q)
    - Check convergence, ESS, consensus tree, etc.
5. **Interpret**
    - Look at posterior clades: do they match known relationships (e.g., suspected plagiarism clusters, known forks)?
    - Look at changes along branches: which features flipped where?

That’s already “Bayesian code phylogenetics” with off-the-shelf machinery, even if the model is naive.

---

8. Where things get more interesting/researchy

Once the basic pipeline works, there are deeper questions:

- **Horizontal transfer & recombination**
    - Code is *much* more recombinatory than DNA. A single repo can mix chunks from many sources.
    - Tree models may be wrong; you might need **phylogenetic networks** or explicit recombination models.
- **Time calibration**
    - In biology we have fossils; here we have timestamps on commits.
    - You can put priors on divergence times informed by commit history or repository creation dates.
- **Edit-level models**
    - Build a generative model over AST edit scripts (insert/delete/move nodes) and use that as your Q.
    - Much closer to what’s *actually* happening during software evolution, but computationally heavy.
- **Coupling with version control graphs**
    - Use Git history as either:

  - a hard constraint on the topology (only estimate internal structure within uncertain areas), or
  - a prior that biases the tree toward the known DAG structure.

---

9. How this connects back to your interest

Conceptually, you're doing exactly what MrBayes does:

- represent objects (programs) as "descriptions" over characters
- posit a stochastic process of change along a hypothetical ancestry tree
- infer the posterior over that ancestry given the observed objects

The tricky part is not the Bayesian machinery; it's **defining the characters and the evolutionary model** so that:

- they're faithful to program evolution, *and*
- simple enough that inference is tractable.

If you want, next step I can:

- sketch an actual **NEXUS data example** for, say, 10 small programs, or
- help design a **feature set** for a concrete use case (e.g., detecting student plagiarism vs. reconstructing the genealogy of classic ELIZA clones), or
- outline what changes you'd need to a MrBayes-like engine to move from "morphological features" to explicit AST edit models.

> How about if we treat the feature problem by asking the LLM to describe each piece of code in high level terms of its algorithm, then use that to descrbe the next one, etc, and round robin this until we settle on a set of features. then use those.

Yeah, that's a really interesting way to attack the "what are the characters?" problem — basically using the LLM as an **automatic feature inventor / annotator**.

Let me translate your idea into something quite concrete and "MrBayes-ready," and then flag the gotchas.

---

1. What you're proposing, in phylo terms

You're saying:

- Instead of hand-designing features (characters),
- Use an LLM to **describe what each program is doing in high-level algorithmic terms**,
- Then iteratively refine those descriptions **across the whole set of programs** until you get a **shared vocabulary of features** (like “uses dynamic programming”, “does BFS”, “approximates with sampling”, etc.),
- Then turn that vocabulary into a **character matrix** and feed it into a Bayesian phylogenetic model.

That’s basically: *unsupervised character discovery* using a language model.

This is totally compatible with the Bayesian machinery; the trick is just to make it structured enough that the output can be treated like proper characters.

---

2. A concrete protocol (how I’d actually do it)

Suppose you have programs $P_1, \ldots, P_n$.

Step 1: First-pass per-program descriptions

For each program, ask the LLM for a **high-level, structured description**. For example:

- main algorithmic strategy
- data structures
- key subroutines / phases
- any noteworthy optimizations or hacks

But critically: **force a schema**, e.g.:

```
text


For this program, list at most 10 atomic features.
Each feature should be a short phrase that could plausibly be true or false of other
implementations of the same task.
Format:
- feature_name: one short phrase
  type: [binary | ordinal]
  description: one sentence
```

Now you’ll have, for each program, its *own* candidate feature list. This is messy and redundant at first, but that’s fine.

Step 2: Round-robin unification of the feature vocabulary

Now do your round-robin idea, but with structure:

1. Start with P_1’s feature list as the initial **global vocabulary**.
2. For each subsequent program P_i:
    - Show the LLM the **current global feature list** (names + brief descriptions).
    - Ask it to:

  - **relabel** P_i in terms of those existing features (set each to 0/1 or categorical value), and
  - optionally propose a *small number* (say ≤ 3) of **truly new features** that are needed to describe P_i and are likely to generalize.

  That prompt is something like:

  > Here is the current feature vocabulary (list). For the new program:
  >
  > 1. For each existing feature, assign a value (0/1/2 etc.).
  > 2. If the program has important algorithmic properties not covered by any existing feature, propose up to 3 new features, in the same schema.

3. When new features are proposed, they're appended to the global vocabulary.

That's your "round robin" discovery: every new program can *reuse* existing characters and *occasionally* introduce new ones.

Step 3: Second pass to ensure consistency

Because feature definitions have evolved as you go, you should do a **second sweep** over all programs with the *final* vocabulary:

- Freeze the global feature list.
- For each program P_i, ask the LLM again:

  > Given the final feature list, assign values to each feature (0/1/2…). Don't invent new features; just classify.

This gives you an $n \times m$ **matrix**: programs × features, where features are aligned across all taxa.

This is now exactly like a **morphological character matrix** for Bayesian phylogenetics.

---

3. Turning the vocabulary into phylo characters

At this point, for each feature:

- Decide its **state space**:
  - binary: 0 = absent, 1 = present
  - ordinal: 0 = none, 1 = low, 2 = medium, 3 = high
  - nominal: 0,1,2 corresponding to discrete alternatives ("strategy A/B/C").
- Encode in NEXUS / MrBayes format as standard or multi-state characters.

Example:

- **`uses_dynamic_programming`** → 0/1
- **`main_graph_traversal`** → states {none, BFS, DFS} → 0/1/2
- **`approximation_strategy`** → {exact, sampling-based, heuristic-greedy}

Then you can:

- use a standard **Mk model** (Lewis) for discrete morphological characters,
- or a mixture if you want different rate categories.

The Bayesian part is entirely standard from there.

4. Benefits of this LLM-based feature discovery

A few nice things about this approach:

1. **You're not guessing the feature set upfront.**
   It's emergent from the diversity of the implementations.
2. Features are **semantic**, not low-level syntactic:
   "uses memoization" is more meaningful than "has a for-loop at token position 37".
3. It's easy to see and interpret clades:
   "this cluster is all the sampling-based approximations,"
   "this cluster is dynamic programming on a grid," etc.
4. It's very compatible with your "hypothesis generation" style:
   the LLM is effectively doing automated qualitative coding, and the Bayesian tree puts structure on those codes.

5. Pitfalls and how to mitigate them

There *are* some important dangers:

(a) Inconsistent labeling across programs

The LLM might decide that "uses DP" is true for one program but miss it in another equivalent one.

Mitigation:

- Always show the **feature definitions** and insist on yes/no/ordinal values only.
- You can also do a **pairwise cross-check**: for pairs of similar programs, ask the LLM to explain any feature differences.

(b) Feature drift over rounds

If you allow the model to redefine or shade meanings of features mid-process, your characters stop being homologous.

Mitigation:

- Treat feature names + descriptions in the global vocabulary as **immutable** once created.
- If something really needs to be split, create *new* features with clearly different definitions and keep the old ones as obsolete or rarely used.

(c) Hallucinated / non-local features

The LLM might invent features that rely on global judgement like “code written by expert” or “highly optimized,” which may not be stable across exemplars.

Mitigation:

- Constrain features to be **operationalizable from the code**: things like data structures, algorithmic approach, patterns of control flow, use of certain APIs.
- Avoid subjective value judgements as characters.

(d) Redundancy and correlation

Many features will be highly correlated (e.g., “uses BFS” and “performs level-order traversal”).

Mitigation:

- You don’t *have* to remove redundancy for Bayesian phylo to work, but you might:
  - prune obvious duplicates manually, or
  - ask the LLM (or a downstream script) to detect almost-always-identical columns and keep one.

---

6. How this still fits the Bayesian view

Notice that all the Bayesian machinery is untouched:

- The **LLM is just a preprocessor** that converts raw code → character matrix.
- Once you have the matrix, the tree inference is “vanilla” Bayesian phylogenetics.

If you want to be conceptually purist, you can think of:

- A full generative story:
  latent tree → latent feature values → code;
  You’re only modeling the middle layer (features) explicitly and treating LLM-based extraction as an approximate inference step for that layer.

In practice, that’s fine; lots of phylo pipelines do messy alignment preprocessing and then treat the alignment as “data”.

---

7. Nice extensions once this is working

If you like this direction, the next steps could be:

- **Two-layer characters**: high-level algorithmic features (via LLM) + low-level AST/metric features.
- **Time calibration**: include commit timestamps or known fork times as calibration points.
- **Ancestral feature reconstruction**: infer what algorithmic strategy earlier “ancestors” had, then ask the LLM to imagine what such a program might look like.

If you want, I can:

- sketch a tiny toy example: 4–5 “programs” with hand-written LLM-style features and show what the resulting NEXUS would look like, or
- help design the *exact* prompts you’d use to drive this feature-discovery loop in an automated pipeline.

Can you construct the prompt that I might use to do the features extraction based on all the above?

Here is a **complete, ready-to-use prompt** that implements the feature-extraction procedure we discussed.
It is designed to be:

- **consistent** across runs,
- **schema-driven**, so features stay aligned,
- **round-robin-compatible**, so new programs can add new features but must reuse existing ones where possible,
- **safe** against drift and hallucinations.

I’ll give you two versions:

1. **First-pass prompt** — when you have *no* global vocabulary yet.
2. **Round-robin prompt** — when you *do* have a global vocabulary and are adding a new program.
3. **Final-classification prompt** — to assign feature values to all programs once the vocabulary is frozen.

You can drop these directly into code.

---

1. FIRST-PASS PROMPT

(Use only for the very first program to bootstrap the vocabulary.)

vbnet

```
You will analyze a piece of source code and extract a set of high-level,
algorithmic features.

Your goal is to propose up to 10 **atomic features** that describe the
algorithmic strategies, data structures, control flow patterns, and
noteworthy implementation techniques of the program.

IMPORTANT RULES:
- Each feature must be something that *other implementations of the same task
  could plausibly either have or not have*.
- Each feature must be **observable from the code** (no subjective judgments).
- Each feature must be **atomic**: not compound or hierarchical.
- Avoid synonyms or overlapping concepts.
- Use short, neutral feature names.
```

```
For each feature, adhere to this schema:

- feature_name: <a short noun-phrase ID, snake_case>
  type: [binary | ordinal | categorical]
  states: <for ordinal/categorical only, list allowed states>
  definition: <1–2 sentence precise definition of what it means for the program
              to have each state>
  value_for_this_program: <one of the states>

Now here is the code to analyze:

<<<CODE>>>

Produce ONLY a YAML list of feature objects, following the schema above.
Do not add commentary or explanation.
```

---

2. ROUND-ROBIN PROMPT (MAIN ONE YOU'LL USE)

(Use for every subsequent program. This keeps features aligned across programs but allows limited expansion of the vocabulary.)

```sql

You will analyze a new program and assign values for an existing global
vocabulary of algorithmic features. You may introduce a very small number
of new features *only if strictly necessary*.

TASK
1. Reuse the existing feature vocabulary.
2. For each existing feature, assign the value for this new program.
3. If the program shows important algorithmic ideas not captured by any
   existing feature, propose at most 3 new features (using the same schema).
4. Be consistent across programs: if two programs appear to implement the
   same concept, they should receive the same feature state.

RULES
- Use only the provided feature names and definitions when assigning values.
  Do NOT reinterpret them.
- New features may be proposed only if they reflect concrete, observable,
  algorithmic aspects lacking from the vocabulary.
- No subjective judgments (e.g., “optimized”, “clean code”).
- Features should be atomic (non-compound) and must be properties that other
  implementations could also have.

SCHEMA (for both existing and new features):

- feature_name: <given or newly proposed>
  type: [binary | ordinal | categorical]
  states: <for ordinal/categorical only>
  definition: <short, precise definition>
  value_for_this_program: <one of the states>

INPUT
Here is the current global feature vocabulary:

<<<FEATURE_VOCABULARY>>>

Here is the program to analyze:

<<<CODE>>>

OUTPUT
Produce a YAML list with:
1. All existing features, with their assigned values.
2. Any newly proposed features, with full schema and values.
```

```
Do not add commentary or explanation.
```

---

3. FINAL CLASSIFICATION PROMPT

(Use once the feature vocabulary is frozen, to ensure consistent assignment across all programs.)

```vbnet
You will classify a program according to a fixed feature vocabulary.
You are NOT allowed to propose new features, modify definitions, or
reinterpret meanings.

For each feature in the vocabulary:
- Read the definition carefully.
- Assign the correct state (binary/ordinal/categorical) based on the code.
- Use 'unknown' if the program does not contain enough evidence to decide.

SCHEMA:

- feature_name: <given>
  value_for_this_program: <state or 'unknown'>

INPUT
Global feature vocabulary (definitions and allowed states):

<<<FEATURE_VOCABULARY>>>

Program to classify:

<<<CODE>>>

OUTPUT
Return ONLY a YAML list of features with assigned values.
```

---

4. Optional: Consistency Check Prompt

(Useful if you want the system to detect inconsistencies before building the matrix.)

```sql
You will check feature assignments for logical or algorithmic consistency.
Given two programs and the feature vocabulary, determine whether any assigned
feature states are inconsistent with the definitions or with each other.

For each inconsistency, produce:
- feature_name
- explanation of the mismatch
- recommended correction (if any)

Do not propose new features.
```

If you want, I can also provide:

- a clean Python script that automates all three stages,
- a format for generating the NEXUS file for MrBayes,
- or an example using a set of 3–5 simple programs to show how the pipeline works end to end.

Assume that I have a directory full of code. Write me the python program that will use ChatGPT's API to implement this. Assume that all the relevant code for each program is in the one file, and that the names of the files are the names of the programs (for the tree). There is an open question about when to stop the round robin. You'll probably have to ask each time whether the LLM made up any new features and stop when an entire round gets "no". What do you think?

Nice, this is a great little “MrBayes front-end” project 😄.
Below is a **single Python script** that:

1. Walks a directory of code files.
2. Uses the OpenAI API to:
    - bootstrap a **feature vocabulary** from the first file,
    - do **round-robin passes** over all files, allowing new features,
    - stop when a full round introduces **no new features** (or a max rounds limit).
3. Does a **final classification pass** with the frozen vocabulary.
4. Writes:
    - `feature_vocab.yaml` — the final global feature set
    - `program_features.csv` — matrix: program × feature value

You’ll need:

- `pip install openai pyyaml`
- `export OPENAI_API_KEY=...`

You can tweak model name, max rounds, etc. at the top or via CLI.

Script: `code_phylo_feature_extractor.py`

python

```python
#!/usr/bin/env python3
import os
import argparse
import time
import csv
import yaml
from typing import Dict, List, Any

from openai import OpenAI

# -------------- CONFIG DEFAULTS --------------

DEFAULT_MODEL = "gpt-5.1"
# Be conservative with rounds; you can increase if needed
DEFAULT_MAX_ROUNDS = 5
```

```
# -------------- PROMPT TEMPLATES --------------

FIRST_PASS_PROMPT = """You will analyze a piece of source code and extract a set of high-level,
algorithmic features.

Your goal is to propose up to 10 atomic features that describe the
algorithmic strategies, data structures, control flow patterns, and
noteworthy implementation techniques of the program.

IMPORTANT RULES:
- Each feature must be something that other implementations of the same task
  could plausibly either have or not have.
- Each feature must be observable from the code (no subjective judgments).
- Each feature must be atomic: not compound or hierarchical.
- Avoid synonyms or overlapping concepts.
- Use short, neutral feature names.

For each feature, adhere to this schema:

- feature_name: <a short noun-phrase ID, snake_case>
  type: [binary | ordinal | categorical]
  states: <for ordinal/categorical only, list allowed states>
  definition: <1–2 sentence precise definition of what it means for the program
              to have each state>
  value_for_this_program: <one of the states>

Now here is the code to analyze:

<<<CODE>>>

Produce ONLY a YAML list of feature objects, following the schema above.
Do not add commentary or explanation.
"""

ROUND_ROBIN_PROMPT = """You will analyze a new program and assign values for an existing global
vocabulary of algorithmic features. You may introduce a very small number
of new features only if strictly necessary.

TASK
1. Reuse the existing feature vocabulary.
2. For each existing feature, assign the value for this new program.
3. If the program shows important algorithmic ideas not captured by any
   existing feature, propose at most 3 new features (using the same schema).
4. Be consistent across programs: if two programs appear to implement the
   same concept, they should receive the same feature state.

RULES
- Use only the provided feature names and definitions when assigning values.
  Do NOT reinterpret them.
- New features may be proposed only if they reflect concrete, observable,
  algorithmic aspects lacking from the vocabulary.
- No subjective judgments (e.g., “optimized”, “clean code”).
- Features should be atomic (non-compound) and must be properties that other
  implementations could also have.

SCHEMA (for both existing and new features):

- feature_name: <given or newly proposed>
  type: [binary | ordinal | categorical]
  states: <for ordinal/categorical only>
  definition: <short, precise definition>
  value_for_this_program: <one of the states>

INPUT
Here is the current global feature vocabulary (YAML):

<<<FEATURE_VOCABULARY>>>

Here is the program to analyze:

<<<CODE>>>

OUTPUT
Produce a YAML list with:
```

```
1. All existing features, with their assigned values.
2. Any newly proposed features, with full schema and values.

Do not add commentary or explanation.
"""

FINAL_CLASS_PROMPT = """You will classify a program according to a fixed feature vocabulary.
You are NOT allowed to propose new features, modify definitions, or
reinterpret meanings.

For each feature in the vocabulary:
- Read the definition carefully.
- Assign the correct state (binary/ordinal/categorical) based on the code.
- Use 'unknown' if the program does not contain enough evidence to decide.

SCHEMA:

- feature_name: <given>
  value_for_this_program: <state or 'unknown'>

INPUT
Global feature vocabulary (definitions and allowed states) in YAML:

<<<FEATURE_VOCABULARY>>>

Program to classify:

<<<CODE>>>

OUTPUT
Return ONLY a YAML list of features with assigned values.
"""


# -------------- OPENAI CLIENT WRAPPER --------------

def call_openai(model: str, prompt: str, temperature: float = 0.0) -> str:
    """
    Simple wrapper around the Chat Completions API.
    """
    client = OpenAI()

    resp = client.chat.completions.create(
        model=model,
        messages=[
            {"role": "system",
             "content": "You are a careful assistant that follows schemas exactly and outputs
valid YAML."},
            {"role": "user", "content": prompt},
        ],
        temperature=temperature,
    )

    return resp.choices[0].message.content


# -------------- UTILITIES --------------

def load_code_files(directory: str) -> Dict[str, str]:
    """
    Load all regular files in the directory as code.
    Program name = filename (without extension).
    """
    programs = {}
    for fname in sorted(os.listdir(directory)):
        path = os.path.join(directory, fname)
        if not os.path.isfile(path):
            continue
        # Skip hidden files
        if fname.startswith("."):
            continue
        prog_name, _ext = os.path.splitext(fname)
        with open(path, "r", encoding="utf-8", errors="ignore") as f:
            code = f.read()
        programs[prog_name] = code
    return programs


def parse_yaml_list(yaml_str: str) -> List[Dict[str, Any]]:
    """
    Parse YAML expected to be a list of feature dicts.
    """
    try:
        data = yaml.safe_load(yaml_str)
        if not isinstance(data, list):
            raise ValueError("YAML output is not a list")
        return data
    except Exception as e:
        raise RuntimeError(f"Failed to parse YAML from model: {e}\n--- RAW ---\n{yaml_str}")


# -------------- MAIN PIPELINE --------------

def bootstrap_vocab_for_first_program(model: str, prog_name: str, code: str) -> List[Dict[str,
Any]]:
    """
    Use FIRST_PASS_PROMPT on the first program to create initial vocabulary.
    """
    prompt = FIRST_PASS_PROMPT.replace("<<<CODE>>>", code)
    print(f"[BOOTSTRAP] Extracting initial features from {prog_name}...")
    yaml_str = call_openai(model, prompt)
    features = parse_yaml_list(yaml_str)
    print(f"[BOOTSTRAP] Got {len(features)} initial features.")
    return features


def round_robin_expand_vocab(
    model: str,
    programs: Dict[str, str],
    max_rounds: int = DEFAULT_MAX_ROUNDS,
) -> List[Dict[str, Any]]:
    """
    Iteratively expand the feature vocabulary by looping over programs.

    Strategy:
      - Round 0: pick the first program, run FIRST_PASS, get initial vocab.
                 Then run ROUND_ROBIN on the remaining programs.
      - Rounds 1..max_rounds-1: for all programs, run ROUND_ROBIN.
      - Track new features per round; stop when a full round adds none.
      - We don't keep per-program values here; final classification is done later.
    """
    prog_items = list(programs.items())
    if not prog_items:
        raise ValueError("No programs found.")

    # --- Round 0: bootstrap with first program ---
    vocab = bootstrap_vocab_for_first_program(model, prog_items[0][0], prog_items[0][1])
    vocab_by_name = {f["feature_name"]: f for f in vocab}

    # Also process remaining programs in round 0 using round-robin prompt
    def run_round(start_index: int, vocab_by_name: Dict[str, Any]) -> int:
        """
        Run a single round of round-robin feature extraction starting at program index
`start_index`.
        Returns the number of new features added in this round.
        """
        new_features_this_round = 0
        for i in range(start_index, len(prog_items)):
            prog_name, code = prog_items[i]
            print(f"[ROUND] Processing {prog_name}...")
            vocab_yaml = yaml.safe_dump(list(vocab_by_name.values()), sort_keys=False)
            prompt = ROUND_ROBIN_PROMPT.replace("<<<FEATURE_VOCABULARY>>>", vocab_yaml)\
                                       .replace("<<<CODE>>>", code)
            yaml_str = call_openai(model, prompt)
            features = parse_yaml_list(yaml_str)

            # Incorporate any new features
            for f in features:
                fname = f.get("feature_name")
                if not fname:
                    continue
                if fname not in vocab_by_name:
                    vocab_by_name[fname] = {
                        "feature_name": fname,
                        "type": f.get("type"),
                        "states": f.get("states"),
                        "definition": f.get("definition"),
                    }
                    new_features_this_round += 1
        return new_features_this_round

    print("[ROUND 0] Expanding vocabulary with remaining programs...")
    new0 = run_round(start_index=1, vocab_by_name=vocab_by_name)
    print(f"[ROUND 0] New features added: {new0}")

    # --- Subsequent rounds ---
    for r in range(1, max_rounds):
        print(f"[ROUND {r}] Starting full round over all programs...")
        new_features = run_round(start_index=0, vocab_by_name=vocab_by_name)
        print(f"[ROUND {r}] New features added: {new_features}")
        if new_features == 0:
            print(f"[STOP] No new features in round {r}; vocabulary converged.")
            break
        if r == max_rounds - 1:
            print("[WARN] Reached max_rounds without convergence; using current vocabulary.")

    # Return vocabulary as a list in a stable order
    final_vocab = list(vocab_by_name.values())
    print(f"[RESULT] Final vocabulary size: {len(final_vocab)}")
    return final_vocab


def final_classification(
    model: str,
    programs: Dict[str, str],
    vocab: List[Dict[str, Any]],
) -> Dict[str, Dict[str, Any]]:
    """
    With vocabulary frozen, classify each program to assign feature values.
    Returns: dict[program_name][feature_name] = value
    """
    feature_matrix: Dict[str, Dict[str, Any]] = {}
    vocab_yaml = yaml.safe_dump(vocab, sort_keys=False)

    for prog_name, code in programs.items():
        print(f"[FINAL] Classifying {prog_name}...")
        prompt = FINAL_CLASS_PROMPT.replace("<<<FEATURE_VOCABULARY>>>", vocab_yaml)\
                                   .replace("<<<CODE>>>", code)
        yaml_str = call_openai(model, prompt)
        features = parse_yaml_list(yaml_str)

        row = {}
        for f in features:
            fname = f.get("feature_name")
            value = f.get("value_for_this_program")
            if fname is None:
                continue
            row[fname] = value
        feature_matrix[prog_name] = row

    return feature_matrix


def write_vocab_yaml(vocab: List[Dict[str, Any]], path: str) -> None:
    with open(path, "w", encoding="utf-8") as f:
        yaml.safe_dump(vocab, f, sort_keys=False)
    print(f"[WRITE] Feature vocabulary written to {path}")


def write_matrix_csv(feature_matrix: Dict[str, Dict[str, Any]], vocab: List[Dict[str, Any]],
path: str) -> None:
    # Make sure all feature columns are present
    feature_names = [f["feature_name"] for f in vocab]

    with open(path, "w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        header = ["program"] + feature_names
        writer.writerow(header)

        for prog_name, row in sorted(feature_matrix.items()):
            values = [row.get(fname, "") for fname in feature_names]
            writer.writerow([prog_name] + values)

    print(f"[WRITE] Program-feature matrix written to {path}")


# -------------- MAIN ENTRY POINT --------------

def main():
    parser = argparse.ArgumentParser(
        description="Iteratively extract algorithmic features from a directory of code using the OpenAI API."
    )
    parser.add_argument("code_dir", help="Directory containing code files (one program per file).")
    parser.add_argument("--model", default=DEFAULT_MODEL, help=f"OpenAI model name (default: {DEFAULT_MODEL})")
    parser.add_argument("--max-rounds", type=int, default=DEFAULT_MAX_ROUNDS,
                        help=f"Maximum number of round-robin rounds (default: {DEFAULT_MAX_ROUNDS})")
    parser.add_argument("--output-prefix", default="code_phylo",
                        help="Prefix for output files (default: code_phylo)")

    args = parser.parse_args()

    programs = load_code_files(args.code_dir)
    if not programs:
        print("No code files found in directory.")
        return

    print(f"Found {len(programs)} programs in {args.code_dir}.")

    # 1) Round-robin vocabulary construction
    vocab = round_robin_expand_vocab(model=args.model, programs=programs, max_rounds=args.max_rounds)

    # 2) Final classification
    feature_matrix = final_classification(model=args.model, programs=programs, vocab=vocab)

    # 3) Write outputs
    vocab_path = f"{args.output_prefix}_feature_vocab.yaml"
    matrix_path = f"{args.output_prefix}_program_features.csv"
    write_vocab_yaml(vocab, vocab_path)
    write_matrix_csv(feature_matrix, vocab, matrix_path)

    print("[DONE]")


if __name__ == "__main__":
    main()
```

How the stopping rule works

- **`round_robin_expand_vocab`** runs **rounds** over all programs.
- In each round, for each program:
    - It sends the current vocabulary + code to the LLM with the **round-robin prompt**.
    - It parses the returned YAML and checks each **`feature_name`**.
    - If a **`feature_name`** isn't in the current **`vocab_by_name`**, it's treated as a **new feature** and added.
- After each full round, if **`new_features == 0`**, it stops — vocabulary is considered converged.

- There's also a `max_rounds` guard to avoid infinite expansion if the model keeps inventing features.

This matches your "stop when an entire round gets no new features" idea, but the detection is done programmatically rather than asking the model to self-report.

If you'd like, I can next:

- add simple **rate limiting / retry** logic,
- or add a function to emit a **NEXUS file** for MrBayes directly from `program_features.csv`.

> A minor (hopefully) change. Tell it to ignore the programming languge in which the program is written, to the extent possible. (Give me the mininmal changes for this.)

Here are the **minimal changes only** — just the lines you should *add* or *modify* in the prompts.
No other parts of the script need to change.

✅ 1. FIRST-PASS PROMPT: minimal change

Add this sentence to the RULES section (anywhere inside the "IMPORTANT RULES" block is fine):

```sql
- Ignore the specific programming language used; extract features that are language-agnostic and algorithmic in nature.
```

✅ 2. ROUND-ROBIN PROMPT: minimal change

Add this line to the RULES block:

```sql
- Ignore the programming language; classify features based only on algorithmic behavior, not syntactic or language-specific constructs.
```

✅ 3. FINAL CLASSIFICATION PROMPT: minimal change

Add this line to the RULES section:

```sql
- Ignore the programming language; evaluate features based solely on algorithmic content, not language-specific syntax.
```

⭐ Exact patches (copy/paste into your code)

Search for each of the three prompt blocks and insert the following lines:

In FIRST_PASS_PROMPT

```python
- Ignore the specific programming language used; extract features that are language-agnostic and
algorithmic in nature.
```

In ROUND_ROBIN_PROMPT

```python
- Ignore the programming language; classify features based only on algorithmic behavior, not
syntactic or language-specific constructs.
```

In FINAL_CLASS_PROMPT

```python
- Ignore the programming language; evaluate features based solely on algorithmic content, not
language-specific syntax.
```

If you want, I can give you a tiny `diff` patch format instead, or directly modify only the relevant strings in the script.

how to I give this my API key? (I'd like it to be a required arg.)

Here is the **minimal, clean way** to make the API key a **required command-line argument** and avoid relying on environment variables.

Below is the **exact patch**, not a full rewrite.

✅ 1. Add a required `--api-key` argument

In your `main()` function, add:

```python
parser.add_argument("--api-key", required=True,
                    help="Your OpenAI API key (required).")
```

✅ 2. Pass the API key into `call_openai()`

Modify the signature of **`call_openai()`**:

```python
def call_openai(api_key: str, model: str, prompt: str, temperature: float = 0.0) -> str:
```

and inside it, replace:

```python
client = OpenAI()
```

with:

```python
client = OpenAI(api_key=api_key)
```

✅ 3. Update every call to `call_openai()`

Search for lines like:

```python
yaml_str = call_openai(model, prompt)
```

and change to:

```python
yaml_str = call_openai(api_key, model, prompt)
```

Specifically:

In `bootstrap_vocab_for_first_program`:

```python
yaml_str = call_openai(api_key, model, prompt)
```

In round-robin loop:

```python
yaml_str = call_openai(api_key, model, prompt)
```

In final classification:

```python
yaml_str = call_openai(api_key, model, prompt)
```

✅ 4. Pass API key from `main()` into helper functions

Modify function signatures to accept **`api_key`**, e.g.:

`bootstrap_vocab_for_first_program`:

```python
def bootstrap_vocab_for_first_program(api_key: str, model: str, prog_name: str, code: str):
```

`round_robin_expand_vocab`:

```python
def round_robin_expand_vocab(api_key: str, model: str, programs: Dict[str, str], max_rounds:
int):
```

`final_classification`:

```python
def final_classification(api_key: str, model: str, programs: Dict[str, str], vocab:
List[Dict[str, Any]]):
```

And update the calls in **`main()`**:

```python
vocab = round_robin_expand_vocab(args.api_key, args.model, programs, args.max_rounds)

feature_matrix = final_classification(args.api_key, args.model, programs, vocab)
```

⭐ Minimal unified diff (copy-pasteable)

If you prefer an actual diff:

```diff
--- a/code_phylo_feature_extractor.py
+++ b/code_phylo_feature_extractor.py
@@ -1,6 +1,7 @@
 def call_openai(
-        model: str, prompt: str, temperature: float = 0.0
+        api_key: str, model: str, prompt: str, temperature: float = 0.0
 ) -> str:
```

```
-     client = OpenAI()
+     client = OpenAI(api_key=api_key)
```

And:

```
diff

--- a/code_phylo_feature_extractor.py
+++ b/code_phylo_feature_extractor.py
@@ -300,6 +301,8 @@ def main():
     parser = argparse.ArgumentParser(...)

+    parser.add_argument("--api-key", required=True,
+                        help="Your OpenAI API key (required).")
```

And every call:

```
diff

- yaml_str = call_openai(model, prompt)
+ yaml_str = call_openai(api_key, model, prompt)
```

If you want...

I can give you the fully updated script with only the required modifications applied — no extraneous changes — so you can drop it in and run it immediately.

Wait, how did it originally get the API key?

Good catch — the original script *did* have a way to get your API key, but it did it **implicitly**, through the **environment variable** that the OpenAI SDK looks for automatically.

✔️ How it originally worked

This line:

```
python

client = OpenAI()
```

uses the OpenAI Python client's default behavior:

- It looks for an environment variable named:
  `OPENAI_API_KEY`
- If found, it uses that.
- If not found — the request will fail.

So originally, you were expected to set:

```bash
export OPENAI_API_KEY="sk-xxxxx"
```

or on macOS:

```bash
export OPENAI_API_KEY=sk-xxxxx
```

or put it in your `.zshrc`, `.bashrc`, etc.

✔️ Why it worked without specifying the key in code

Because the library automatically checked the environment variable.

✔️ Why adding a required argument changes behavior

Once you make `--api-key` a required argument and pass it explicitly to:

```python
client = OpenAI(api_key=api_key)
```

you now **override** the environment mechanism and must supply the key via CLI.

This is safer and more transparent, but requires the changes I gave you.

⭐ TL;DR

Originally:

OpenAI key was read from the environment:

```ini
OPENAI_API_KEY=sk-xxxx python script.py ./code_dir
```

After your requested change:

The script will *require*:

```bash
python script.py ./code_dir --api-key sk-xxxx
```

If you'd like, I can give you:

- a version that supports **either** environment variable **or** CLI,
- a version that warns if the environment key is missing,
- or a version that reads from a `.env` file using `python-dotenv`.

there's no yaml package in python did you mean PyYAML?

Yes — exactly.
The correct package is **PyYAML**, but in Python you import it as:

```python
import yaml
```

So:

- The *library name on pip* is: **PyYAML**
- The *module name you import* is: **yaml**

This is just one of those legacy mismatches (like `pip install pillow` → `from PIL import Image`).

✔️ To install it:

```bash
pip install pyyaml
```

Then this works:

```python
import yaml
data = yaml.safe_load("x: 1")
```

✔️ The script I gave you expects *PyYAML* (importing as `yaml`)

So nothing else changes — just install PyYAML and you're good.

If you want, I can add a graceful error message in the script that says:

```csharp
PyYAML is required. Install with: pip install pyyaml
```

Just say the word.

```
Traceback (most recent call last):
  File
"/Users/jeffshrager/Desktop/AIHistory/ELIZA_See_also_GDrive/elizagen_repo/genealogy/elizagen.p
y", line 370, in <module>
    main()
  File
"/Users/jeffshrager/Desktop/AIHistory/ELIZA_See_also_GDrive/elizagen_repo/genealogy/elizagen.p
y", line 355, in main
    vocab = round_robin_expand_vocab(model=args.model, programs=programs,
max_rounds=args.max_rounds)

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^
  File
"/Users/jeffshrager/Desktop/AIHistory/ELIZA_See_also_GDrive/elizagen_repo/genealogy/elizagen.p
y", line 222, in round_robin_expa\
nd_vocab
    vocab = bootstrap_vocab_for_first_program(model, prog_items[0][0], prog_items[0][1])

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File
"/Users/jeffshrager/Desktop/AIHistory/ELIZA_See_also_GDrive/elizagen_repo/genealogy/elizagen.p
y", line 196, in bootstrap_vocab_\
for_first_program
    yaml_str = call_openai(model, prompt)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: call_openai() missing 1 required positional argument: 'prompt'
```

Ah, this traceback is very informative — it’s telling us exactly where the signature got mangled.

```
TypeError: call_openai() missing 1 required positional argument: 'prompt'
```

This almost always means: **there’s a required parameter (`prompt`) that comes *after* a parameter with a default (`temperature`) in the function definition.** Python doesn’t allow that ordering, and it also messes up how your arguments are bound.

Right now your code almost certainly has something like:

```python
def call_openai(model: str, temperature: float = 0.0, prompt: str) -> str:
    ...
```

Then you call it as:

```
python

yaml_str = call_openai(model, prompt)
```

What happens:

- **`model`** → **`model`**
- **`prompt`** → **gets bound to `temperature`**
- **`prompt`** (the parameter) is left unbound → **`TypeError: missing 1 required positional argument: 'prompt'`**.

Fix: put `prompt` before any defaulted params

You want **`prompt`** to be the *second* parameter, before anything with a default.

If you are not yet passing an API key

Make sure your definition looks exactly like this:

```
python

def call_openai(model: str, prompt: str, temperature: float = 0.0) -> str:
    """
    Simple wrapper around the Chat Completions API.
    """
    client = OpenAI()

    resp = client.chat.completions.create(
        model=model,
        messages=[
            {"role": "system",
             "content": "You are a careful assistant that follows schemas exactly and outputs
valid YAML."},
            {"role": "user", "content": prompt},
        ],
        temperature=temperature,
    )

    return resp.choices[0].message.content
```

Then calls like:

```
python

yaml_str = call_openai(model, prompt)
```

are fine.

If you *did* already add the `api_key` argument

Then the correct signature should be:

```python
def call_openai(api_key: str, model: str, prompt: str, temperature: float = 0.0) -> str:
    client = OpenAI(api_key=api_key)
    ...
```

and all calls should be:

```python
yaml_str = call_openai(api_key, model, prompt)
```

But the same rule holds: all **required** arguments (`api_key`, `model`, `prompt`) must come **before** the one with a default (`temperature`).

Minimal checklist for you

1. Find your `call_openai` definition.
2. Make sure the parameters are in this order:
    - without explicit key: `model, prompt, temperature=0.0`
    - with explicit key: `api_key, model, prompt, temperature=0.0`
3. Don't put `prompt` after `temperature` in the signature.
4. Leave your calls as `call_openai(model, prompt)` (or `call_openai(api_key, model, prompt)`).

Once you fix that ordering, this TypeError should go away.

Wait -- can't you see your own code?

Short answer: no, I can't see your actual file on disk. 😄

I only "see":

- The code **I previously sent you in this chat**, and
- Whatever snippets or errors **you paste back** (like that traceback).

Once you copy my code into `elizagen.py` and start editing, I don't automatically see those edits unless you paste the relevant parts here. So when I debug, I'm reasoning from:

- The original version I gave you
- The error message / stack trace
- What *probably* changed in between

That's why I sometimes talk in terms of "you probably have something like..." — I'm inferring, not literally reading your file.

If you'd like me to be very precise about this **`call_openai`** issue, paste just:

- the current **`def call_openai(...)`** definition, and
- one of the lines where you call it (e.g., the one in **`bootstrap_vocab_for_first_program`**),

and I can tell you exactly what to change, line by line.

I didn't make any changes, except the name.

Ah — if you *didn't touch anything except the filename*, then the error **cannot be explained from the original code I gave you**, because the original version has the correct argument order:

```python
def call_openai(model: str, prompt: str, temperature: float = 0.0)
```

and this safely supports:

```python
yaml_str = call_openai(model, prompt)
```

So the only way Python can produce:

```css
TypeError: call_openai() missing 1 required positional argument: 'prompt'
```

is if the **function definition in your file is no longer in that order**.

This *definitely* means one of these is true:

1. Your **`call_openai`** now has **`temperature`** before **`prompt`**.
2. Or a copy/paste shifted parameter positions.
3. Or your editor wrapped a long line and rearranged arguments.
4. Or a merge or find/replace accidentally changed it.
5. Or the script is importing a different **`call_openai`** from somewhere else (less likely).

There's no other mechanism in Python that produces that specific error.

✔️ Let's check directly

Please paste **exactly these two things** from your file:

1. **Your current definition of `call_openai`**, starting with:

```
python

def call_openai(
```

through its closing parenthesis/colon.

2. **One of the lines where it is called**, e.g.:

```
python

yaml_str = call_openai(model, prompt)
```

Once I see those two small snippets, I can tell you with 100% precision exactly what's wrong.

(And don't worry — this kind of parameter-order bug is extremely common and usually a 1–2 line fix.)

Here's the call:
def call_openai(api_key: str, model: str, prompt: str, temperature: float = 0.0) -> str:

Great, this explains the error perfectly.

You now have:

```
python

def call_openai(api_key: str, model: str, prompt: str, temperature: float = 0.0) -> str:
```

but your *calls* almost certainly still look like:

```
python

yaml_str = call_openai(model, prompt)
```

Here's how Python binds that:

- **`api_key`** ← **`model`**
- **`model`** ← **`prompt`**
- **`prompt`** ← **missing** ❌

So Python complains: **`missing 1 required positional argument: 'prompt'`**.

Nothing mystical — it's just argument order.

Minimal fix

1. **Leave your definition as-is** (it’s fine):

```python
def call_openai(api_key: str, model: str, prompt: str, temperature: float = 0.0) -> str:
    client = OpenAI(api_key=api_key)
    ...
```

2. **Update every call** to pass the API key as the first argument.

Anywhere you currently have something like:

```python
yaml_str = call_openai(model, prompt)
```

change it to:

```python
yaml_str = call_openai(api_key, model, prompt)
```

Concretely, that means:

In `bootstrap_vocab_for_first_program`:

Change the function signature to accept `api_key`:

```python
def bootstrap_vocab_for_first_program(api_key: str, model: str, prog_name: str, code: str) -> 
List[Dict[str, Any]]:
```

and change:

```python
yaml_str = call_openai(model, prompt)
```

to:

```python
yaml_str = call_openai(api_key, model, prompt)
```

In `round_robin_expand_vocab`:

Signature:

```python
def round_robin_expand_vocab(
    api_key: str,
    model: str,
    programs: Dict[str, str],
    max_rounds: int = DEFAULT_MAX_ROUNDS,
) -> List[Dict[str, Any]]:
```

Inside **`run_round`**, change:

```
python

yaml_str = call_openai(model, prompt)
```

to:

```
python

yaml_str = call_openai(api_key, model, prompt)
```

In `final_classification`:

Signature:

```
python

def final_classification(
    api_key: str,
    model: str,
    programs: Dict[str, str],
    vocab: List[Dict[str, Any]],
) -> Dict[str, Dict[str, Any]]:
```

and again:

```
python

yaml_str = call_openai(api_key, model, prompt)
```

In `main()`:

Pass **`args.api_key`** down:

```
python

vocab = round_robin_expand_vocab(
    api_key=args.api_key,
    model=args.model,
    programs=programs,
    max_rounds=args.max_rounds,
)

feature_matrix = final_classification(
    api_key=args.api_key,
    model=args.model,
    programs=programs,
    vocab=vocab,
)
```

Once every call is **`call_openai(api_key, model, prompt)`**, that “missing prompt” error should disappear.

If I undo the change to require the API key and just use the env var will this eliminate this mess?

Yes.
If you revert to the original pattern (environment variable + original function signature), *all of this disappears immediately*.

Here's exactly what to do for the cleanest rollback:

✅ 1. Restore the original function signature

Change:

```python
def call_openai(api_key: str, model: str, prompt: str, temperature: float = 0.0):
```

back to:

```python
def call_openai(model: str, prompt: str, temperature: float = 0.0):
```

And restore:

```python
client = OpenAI()
```

(no api_key parameter)

Because the OpenAI Python client will automatically use:

```nginx
OPENAI_API_KEY
```

from your environment.

✅ 2. Remove the added `api_key` argument everywhere

Remove:

```python
--api-key
```

from your CLI options.

Remove `api_key=` from helper functions.

Remove `api_key` from the argument list of:

- `bootstrap_vocab_for_first_program`
- `round_robin_expand_vocab`
- `final_classification`

And remove `api_key` from the calls to these.

✅ 3. Ensure your environment variable is set

In Bash/zsh:

```bash
export OPENAI_API_KEY="sk-..."
```

Verify:

```bash
echo $OPENAI_API_KEY
```

✨ After doing this, everything returns to the original working state.

No reordering errors, no parameter mismatches.

Why this eliminates the mess

The entire cascade of bugs came from:

1. Adding an extra argument (`api_key`)
2. Some function calls not updated
3. Parameter-ordering mismatches with default values
4. Python binding positional arguments incorrectly

When you revert to the environment variable pattern:

- No added args
- No data flow changes
- No call-site updates

- No signature complexity
- No position-sensitive issues

It becomes simple and robust.

If you'd like...

I can generate a **clean copy of the script**, fully updated with all the feature extraction logic *but without* the API-key argument — fresh and consistent end-to-end.

Just say **"give me the clean version"** and I'll produce it.

okay, so I'm on a mac which dot file do I put the key into

On macOS (using the default **zsh** shell), you almost always want to put your **`OPENAI_API_KEY`** into:

✅ `~/.zshrc`

This is the standard user shell startup file for terminal sessions.

✔️ To add your API key

1. Open your zsh config:

```bash
nano ~/.zshrc
```

2. Add this line at the bottom:

```bash
export OPENAI_API_KEY="sk-xxxxxxxxxxxxxxxx"
```

3. Save and exit (**`Ctrl+O`**, **`Enter`**, **`Ctrl+X`**).
4. Reload your shell:

```bash
source ~/.zshrc
```

5. Verify:

```bash
echo $OPENAI_API_KEY
```

You should see your key.

❓ What about `.bashrc`, `.bash_profile`, `.profile`, etc.?

Since macOS Catalina (10.15), the **default shell is zsh**, not bash.
Therefore:

- `.bashrc` — ignored unless you explicitly run bash
- `.bash_profile` — ignored for zsh
- `.profile` — rarely used
- `.zshenv` — loads for all shells (but too early; not recommended for secrets)
- `.zprofile` — login shells (still works, but `.zshrc` is simpler)

So the correct place is `~/.zshrc`.

⭐ TL;DR

Put your key here:

👉 `~/.zshrc`

with:

```bash
export OPENAI_API_KEY="sk-xxxx"
```

and you're done.

If you'd like, I can give you a one-liner that both appends the key to `.zshrc` and reloads the shell automatically.