Energy consumption of a Web Application Implemented in Different Programming Languages and Web Frameworks

Probabilistic Programming 2025 Exam by Raúl Pardo (raup@itu.dk) and Andrzej Wąsowski (wasowski@itu.dk)
version 1.0.0 2025-03-20 08:40

In this exam, your task is to analyze energy consumption of different implementations of a web application. The goal is to determine whether there are differences in energy consumption in different implementations of the web application or in its API endpoints. This analysis is of utmost importance, as it might help software engineers to make informed choices that lower energy consumption. For instance, a plausible hypothesis is that lower level programming languages such as Rust consume less energy than higher level languages such as Python. A preconception in this domain is that running time is the driving factor in energy consumption. Are these true? The data in this exam and the analysis you will develop will allow to answer this type of questions.

Data

The dataset contains $N = 1960$ measurements of energy consumption for different implementations and functionality of a web application. For each setup, there are 20 measurements. The dataset is in the file dataset.csv. The variables in the dataset are:

- **Application**. This variable has the form `<programming_language>-<web_framework>`. It specifies the programming language and web framework used in the experiment. For instance, `rust-actix` denotes the web framework Actix for the programming language Rust, or `c-sharp-razor` denotes the web framework Razor for the programming language C#.
    - Note that if a hypothesis involves only programming language, you need to extract it from the values in this variable.
- **Endpoint**. This variable refers to the API endpoints of the web application. For example, `/api/register` refers to the API endpoint used for registering users in the web application, or `/logout` is used for logging out of the system.
- **Runtime**. This variable indicates the time it took to process the request to the endpoint in seconds.
- **Energy consumption**. This variable indicates the energy consumed for processing the request to the endpoint in Joules.

Each row in the dataset is a measurement of the total energy consumed and runtime after processing a request in the corresponding API endpoint. The *Application* variable in each row indicates the web framework used for the measurement.

Hypotheses

To analyze energy consumption in the different implementations, you must investigate the following hypotheses:

- **H1** - The web framework `c-sharp-razor` consumes more energy than any other web framework in the dataset.
- **H2** - The programming language `javascript` consumes the least amount of energy compared to any other programming language in the dataset.
- **H3** - Runtime has a stronger impact on energy consumption for some API endpoints than others. That is, the effect of runtime on energy consumption is larger for some API endpoints than others.

Your task is to use Bayesian Inference and Regression to decide whether these hypotheses hold, or possibly reject them. This includes:

- Loading, restructuring and transforming the data as needed.
- Designing Bayesian regression models and using inference algorithms to test the above hypotheses in PyMC.
- Explaining your model idea in English, preferably using a figure, and showing the Python code.
- Checking and reflecting (in writing) on the quality of the sampling process, considering warnings from the tool, sampling summary statistics, trace plots, and autocorrelation plots. Comment whether the quality of the sampled trace is good, and whether you had to make any adjustments during modeling and/or sampling.
- Visualizing the posterior information appropriately to address the hypotheses.

You should hand in a zip file with a Jupyter notebook and the data file (so that we can run it), and a **PDF file rendering of the Jupyter notebook**, so that your work can be assessed just by reading this file. It appears that the best PDF rendering is obtained by File / Export to HTML, and then saving/printing to PDF from your browser.

Make sure the notebook is actually a **report** readable to the examiners, especially to the censor who has not followed the course. The report should include:

- A brief introduction.
- Explanations on how data is loaded and cleaned.
- Explanations on analysis and model design (for each of the models you consider).
- A discussion of sampling quality (for each model) and all the plots that you present, and a reflection/decision on the outcome for each hypothesis.
- An overall conclusion.

IMPORTANT: For the tasks below, your code must accompany an explanation of its meaning and intended purpose. **Source code alone is not self-explanatory**. As mentioned above, you should also reflect on the results you get, e.g., highlighting issues with the data, or issues, pitfalls and assumptions of a model. **Exams containing only source code or very scarce explanations will result in low grades, including failing grades.**

Minimum requirements

1. Design a regression model to predict energy consumption using web framework as a predictor.
2. Analyze hypothesis H1 using the regression model in (1.).

Ideas for extension

Groups aiming at grade 7 and more should complete the following tasks:

1. Analyze hypothesis H2, if necessary design a new model.
2. Perform prior predictive checks in all your models. Explain why the priors you selected are appropriate.
3. Perform posterior predictive checks in all your models. Discuss the results in the posterior predictive checks.
4. Discuss trace convergence in all your models.

Groups aiming at grade 10 and higher should try 3-5 ideas from below or add some of your own:

1. Analyze hypothesis H3, if necessary design a new model.
2. Perform a counterfactual analysis in your model for H3: For each endpoint, plot posterior predictions on energy consumption for a runtime value much larger than those in the dataset. Does this affect/introduce differences between energy consumption for different endpoints?
3. Design models with a transformation of the predicted variable, i.e., energy consumption. For instance,
    - Build a model to analyze the probability that the energy consumption of a web framework is below 0.4 Joules. You may consider versions of this task involving other predictors.
    - Transform energy consumption into an ordinal variable representing an energy mark, e.g., an energy consumption in 0.0-0.2 is energy mark A, energy consumption in 0.2-0.4 is energy mark B and energy consumption of ≥ 0.4 is energy mark C. Use an ordinal regression model to analyze the energy mark of each framework. You may consider versions of this task involving other predictors.

4. Use information criteria to compare the models to analyze H1, H2 and H3.
5. Design a meaningful multilevel model in the context of these data.
6. Use causal reasoning to analyze causal relations between the variables in the dataset.

SOLUTION

Introduction

This report investigates the energy consumption of a web application implemented using different programming languages and web frameworks. The motivation is to assess whether certain implementations are more energy-efficient, with the broader goal of helping software engineers make sustainable design choices.

Using a dataset of 1,960 energy measurements collected across multiple combinations of programming languages, web frameworks, and API endpoints, we apply Bayesian statistical modeling to examine three main hypotheses:

---

1. **H1**: The web framework `c-sharp-razor` consumes more energy than any other web framework.
2. **H2**: The programming language `javascript` consumes the least amount of energy compared to all other languages.
3. **H3**: Runtime has a stronger impact on energy consumption for some API endpoints than others. That is, the effect of runtime on energy consumption is larger for some API endpoints than others.

We employ regression modeling in `PyMC`, following the Bayesian inference techniques taught in the *Statistical Rethinking* course by Richard McElreath. This includes careful model specification, prior and posterior predictive checks, diagnostics of sampling quality (e.g., trace plots, R-hat, effective sample sizes), and interpretation of posterior distributions.

The analysis involves data transformation (e.g., separating language and framework), encoding categorical predictors, and comparing models based on their ability to support or refute the hypotheses. Results are presented visually and numerically, with reflections on modeling assumptions and inference quality.

Data Cleaning and Preparation

Imports

In [1]:
```
### IMPORTS ###
import numpy as np
import pandas as pd
import arviz as az
import pymc as pm
import matplotlib.pyplot as plt
from causalgraphicalmodels import CausalGraphicalModel
import graphviz
import seaborn as sns
az.style.use("arviz-darkgrid")
```

Data

We load the dataset directly into a `pandas` DataFrame using the `pandas.read_csv` function.

We extract the columns `language` and `framework` from the `application` column, using a regular expression pattern to split the `application` into on dashes, although ensuring to keep the `c-sharp` language intact.

In [2]:
```
# Load the # Load the dataset
df = pd.read_csv('dataset.csv')

# Extract language & framework name, and giving them a column
each (for later use)
df[['language', 'framework']] =
df['application'].str.extract(r'([^-]+(?:-sharp)?)-(.+)')
```

Initial Causality Analysis

We want to perform an intial analysis of the data to understand the relationships between variables.

Assumptions

1. **Observational Data:** The `dataset` contains observational data, i.e., not generated from a controllod experiment. Thus, while we are able to identify *correlations* and *potential influences*, cannot definitively prove any *causal relationships*.
2. **Outcome Variable:** Out primary outcome variable across all three hypotheses is `energy_consumption`, and we aim to understand whiich factors influence it.
3. **Independent Variables/Predictors:**
   - `application` (and its derived features `language` and `framework`):
     - For H1, `framework` is assumed to be a potential cause of `energy_consumption`.
     - For H2, `language` is assumed to be a potential cause of `energy_consumption`.
   - `endpoint`:
     - For H3, `endpoint` is assumed to be a *moderator* of the relationship between `runtime` and `energy_consumption` (i.e., the effect of `runtime` on energy consumption might vary by `endpoint`).
   - `runtime`:
     - For H3, `runtime` is assumed to be a potential cause of `energy_consumption`.

- For broader context, `runtime` could also be assumed to be an intermediate outcome influenced by `application` and `endpoint`.

4. **Assumed Direction of Causality:**
   - For H1 and H2: `language` / `framework` -> `energy_consumption`.
   - For H3: `runtime` -> `energy_consumption`, moderated by `endpoint`. `endpoint` also influences `runtime`.

Impact Analysis

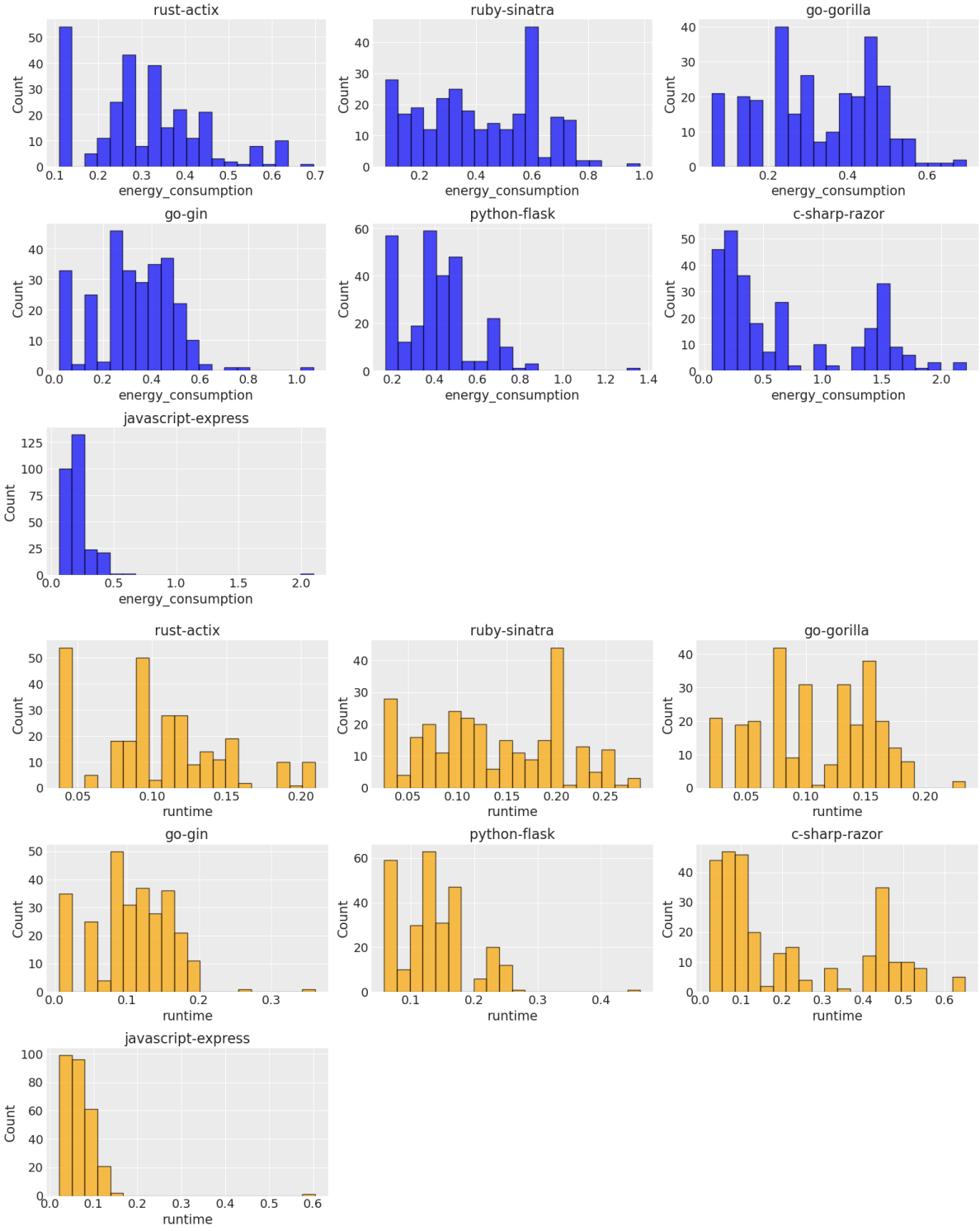
We explore the impact of independent variables on energy consumption and runtime by looking at the distributions of the predictors, grouped by `application`.

We start with the impact of `application` on `energy_consumption` and `runtime`:

In [3]:
```
# Generate subplots of energy consumption distributions, one for each application
for outcome, color in zip(('energy_consumption', 'runtime'), ('blue', 'orange')):

    fig, axes = plt.subplots(3, 3, figsize = (16, 10))
    unique_apps = df['application'].unique()
    num_apps = len(unique_apps)
    max_plots = 3 * 3 # Total number of subplots in the grid

    for i, app in enumerate(unique_apps):
        # Get the data for the current application
        app_outcome = df[df['application'] == app][outcome]

        # Create a histogram of distribution
        ax = axes[i // 3, i % 3]
        ax.hist(app_outcome, bins = 20, alpha = .7, color = color, edgecolor = 'black')

        # Set title, labels
        ax.set_title(app)
        ax.set_xlabel(outcome)
        ax.set_ylabel('Count')

    # Hide unused subplots
```

```
    for i in range(num_apps, max_plots):
        axes[i // 3, i % 3].axis('off')

    # Layout
    plt.tight_layout()
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/274743941
8.py:27: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



Note: the x-axes on the above plots are not aligned.

Interpretation of the impact of `Language` and `Framework` (`Application`) on **Energy Consumption** and **Runtime**:

- **Hypothesis:** The choice of application (language and framework) significantly influences energy consumption and runtime.
- **Observations:**
  - **Low Energy Consumption/Runtime:**

    The `javascript-express` application generally seems to have very low energy consumption and runtime, although with at least one significant energy consumption outlier at just above 2 joules, and one runtime outlier at around 0.6 second (these two outliers might very well be the same if energy consumption and runtime are correlated). The effect of the language `javascript` on energy consumption is further explored in H2.

    `rust-actix` also seems to be consistently effective in both energy consumption and runtime, with most observations below 0.5 joules and 0.2 seconds, respectively.

    The two `go`-based applications (`go-gin` and `go-echo`) likewise seem to be effective, comparable to or slightly higher than `rust-actix`, but better than interpreted languages `python` and `ruby`.

  - **High Energy Consumption/Runtime:**

    The `c-sharp-razor` application generally seems to have the highest energy consumtion, with a significant amount of obversations above 1 joule. The effect of the framework `razor` on energy consumption is further explored in H1.

    `ruby-sinatra` also tends to consume more energy and have longer runtimes when compard to `rust`, `go`, or `javascript` applications, but are generally better than `c-sharp` and `ruby`.

    `python-flask` has moderate to high energy consumption and runtime, generally perfoming worse than `rust`, `go`, and `javascript` applications.

  - **Potential Causal Implications:**

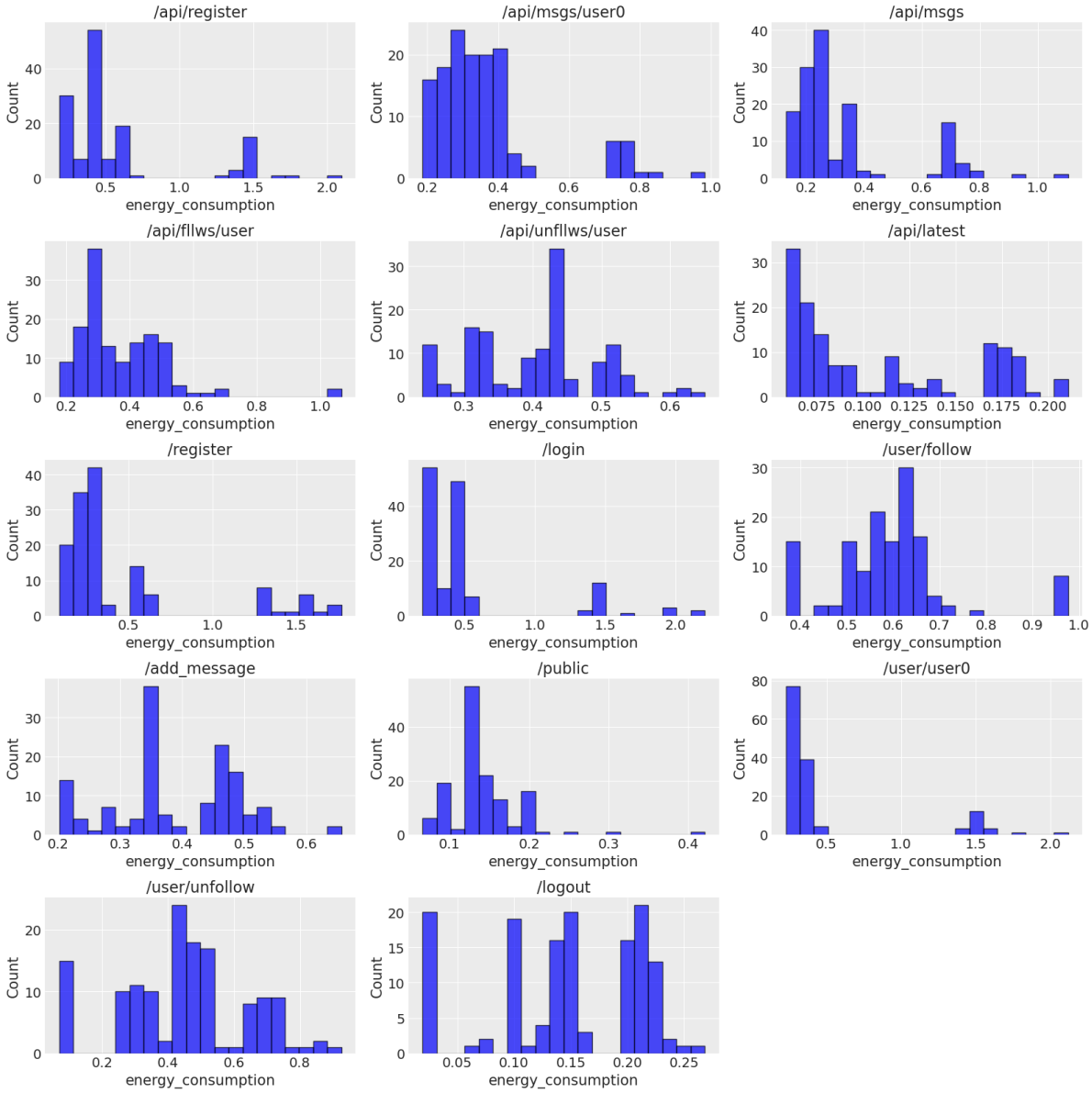
    Compiled languages (`rust`, `go`) generally have efficient rumtimes, which might be a leading effect of more efficient energy consumption. `javascript-express` seems the most efficient, but has at least one significant outlier.

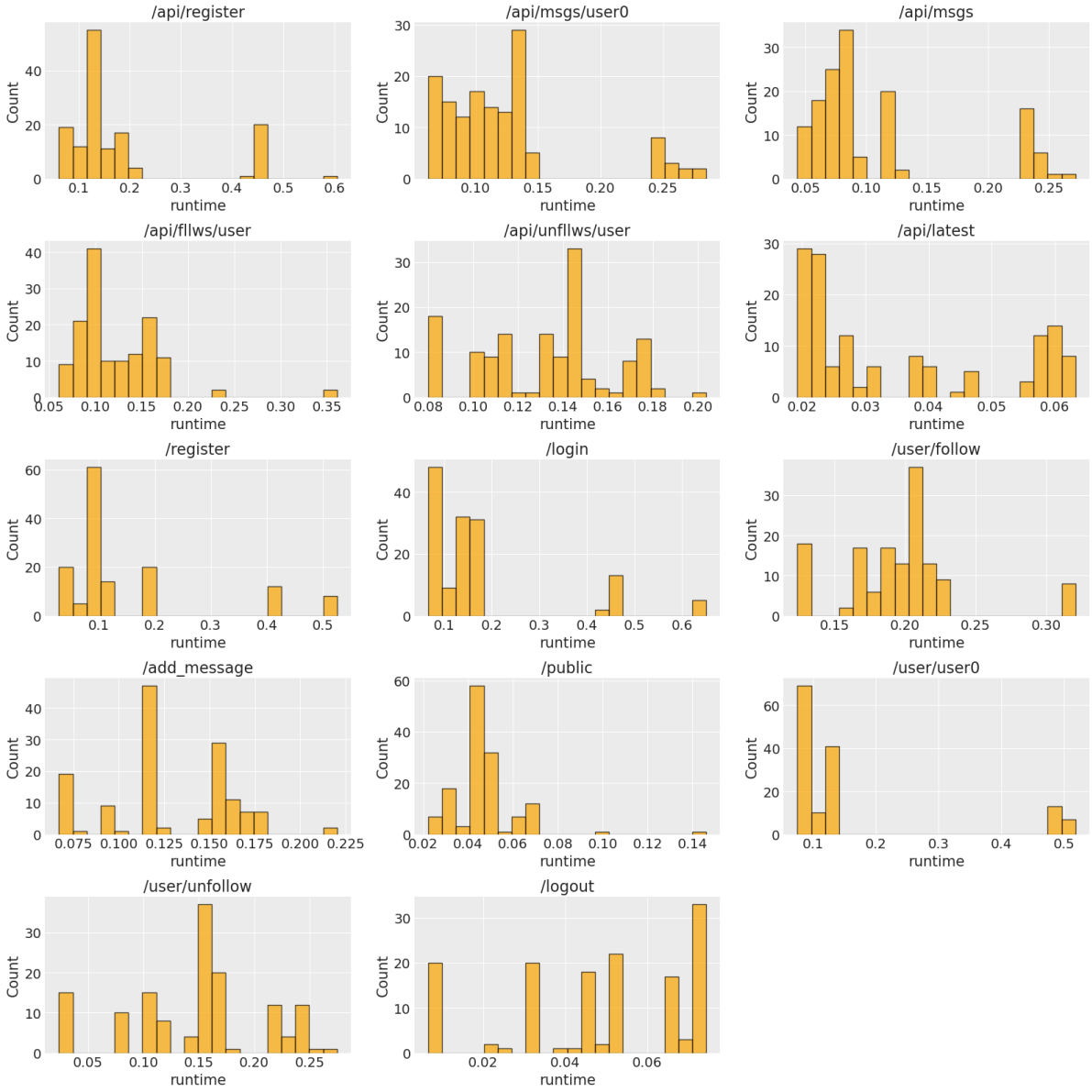
Next, we analyze the impact of `endpoint` on `energy_consumption` and `runtime`:

In [4]:
```
# Generate subplots of energy consumption distributions, one for each endpoint
```

```
for outcome, color in zip(('energy_consumption', 'runtime'),
('blue', 'orange')):

    fig, axes = plt.subplots(5, 3, figsize = (16, 16))
    unique_endpoints = df['endpoint'].unique()
    num_endpoints = len(unique_endpoints)
    max_plots = 5 * 3 # Total number of subplots in the grid

    for i, endpoint in enumerate(unique_endpoints):
        # Get the data for the current endpoint
        endpoint_outcome = df[df['endpoint'] == endpoint]
[outcome]

        # Create a histogram of distribution
        ax = axes[i // 3, i % 3]
        ax.hist(endpoint_outcome, bins = 20, alpha = .7, color =
color, edgecolor = 'black')

        # Set title, labels
        ax.set_title(endpoint)
        ax.set_xlabel(outcome)
        ax.set_ylabel('Count')

    # Hide unused subplots
    for i in range(num_endpoints, max_plots):
        axes[i // 3, i % 3].axis('off')

    # Layout
    plt.tight_layout()
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/825117254.
py:27: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```


/api/register
/api/msgs/user0
/api/msgs
/api/fllws/user
/api/unfllws/user
/api/latest
/register
/login
/user/follow
/add_message
/public
/user/user0
/user/unfollow
/logout
Count
energy_consumption



Note: the x-axes on the above plots are not aligned.

Interpretation of the impact of `endpoint` on **Energy Consumption** and **Runtime**:

- **Hypothesis:** The choice of endpoint significantly influences energy consumption and runtime. I.e., more complex operations, such as database interactions or complex calculations, would consume more resources.
- **Observations:**

  Endpoints like `/logout`, `/api/latest`, and `/public` generally are most efficient in both energy consumption and runtime, suggesting that these are likely simple operations, such as fetching a small payload or status.

  Endpoints like `/api/register`, `/api/login`, and `/user/follow` tend to consume more energy and have longer runtimes, as these potentially invole database write/reads, user authentication, or larger data retrievals.

- **Potential Causal Implications:**

  Endpoint workloads are likely highly influencial to energy consumption and runtime. Simpler tasks use less energy and has shorter runtimes.

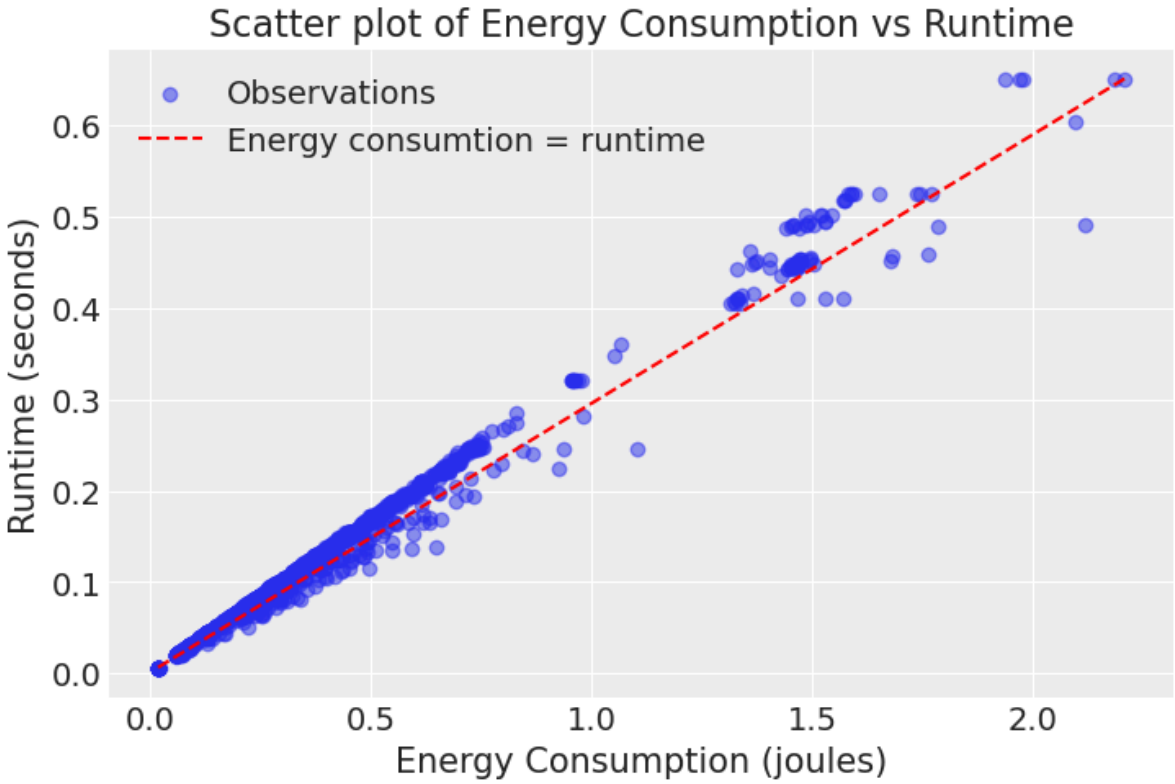
Relationship between Energy Consumption and Runtime

We suspect that energy consumption and runtime are positively correlated. To investigate this, we calculate the correlation coefficient and show `energy_consumption` vs `runtime` on a scatter plot.

In [5]:
```
# Compute correlation coefficient
correlation = df['energy_consumption'].corr(df['runtime'])
print(f"Correlation between energy consumption and runtime:
{correlation:.4f}")

plt.scatter(df['energy_consumption'], df['runtime'], alpha = .5)
plt.plot([df['energy_consumption'].min(),
df['energy_consumption'].max()],
         [df['runtime'].min(), df['runtime'].max()], color =
'red', linestyle = '--')
plt.xlabel('Energy Consumption (joules)')
plt.ylabel('Runtime (seconds)')
plt.title('Scatter plot of Energy Consumption vs Runtime')
plt.legend(['Observations', 'Energy consumtion = runtime'])
```

```
Correlation between energy consumption and runtime: 0.9933
```

Out[5]:
```
<matplotlib.legend.Legend at 0x16dc2ed80>
```



We find that the correlation coefficient between energy consumption and runtime is approximately 0.9933, indicating a nearly perfect positive correlation. This suggests that as runtime increases, energy consumption also tends to increase. However, we also find a few noticable outliers in the plot, indicating that there might be some cases where energy consumption is high despite lower runtime, or vice versa. This could be due to different factors, such as code implementation or hardware differences.

Hypothesis H1

* H1 - The web framework `c-sharp-razor` consumes more energy than any other web framework in the dataset.

To evaluate hypothesis **H1** we design a Bayesian regression model, predicting energy consumption based on framework.

We are assuming a **linear model**, specified as follows:

$$
\begin{aligned}
h_i &\sim \mathcal{N}(\mu_i, \sigma) && [\text{likelihood for observed outcome } h_i] \\
\mu_i &= \alpha + \beta_{F[i]} && [\text{linear model } x \text{ predictor}]
\end{aligned}
$$

The likelihood assumes that the observed energy consumption values h_i are normally distributed around the predicted mean μ_i with a constant standard deviation σ.

The mean μ_i is modeled as a linear combination of the intercept α and the framework effects β.

The prior probailities for parameters α, β, and σ are chosen based on intution and domain knowledge, as explained later.

The modeling process then follows the below steps:

1. Model visualization
2. Data preparation
3. Prior probability selection
4. Prior predictive checks
5. Model fitting and sampling

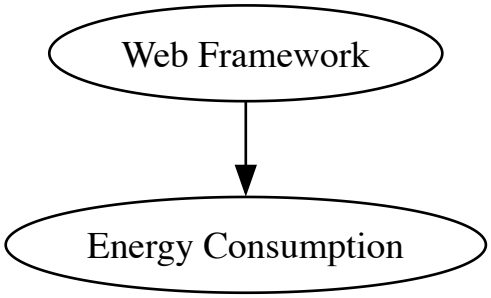
6. Trace plot and summary analysis
7. Posterior predictive checks
8. Hypothesis conclusion

1. Model Visualization

First, we draw the model design as a simple directed acyclic graph (DAG).

In [6]:
```
# Initializing a Causal Inference Model (DAG: Directed Acyclic
Graph)
CausalGraphicalModel(
    nodes=["Web Framework", "Energy Consumption"],
    edges=[
        ("Web Framework", "Energy Consumption"),
    ]
).draw()
```

Out[6]:



2. Data Preparation

Creating dummy variables for the categorical variable `framework` allows us to use them as individual predictors in the regression model. Since we hypothesize that `c-sharp-razor` consumes the most energy, we use this framework as the *baseline* category. As such, we exclude its dummy variable from the regression model. This means the coefficients of other frameworks will be interpreted relative to `c-sharp-razor`.

In [7]:
```
# Get framework categories
df['framework'] = df['framework'].astype('category')
current_categories = df['framework'].cat.categories.tolist()

# Set razor as first category and drop it
df['framework'] = df['framework'].cat.reorder_categories(
    ['razor'] + [cat for cat in current_categories if cat !=
```

```
'razor'],
    ordered = False
)

# Combine with original data
X_h1 = pd.get_dummies(df['framework'], drop_first = True)
y = df['energy_consumption'].values
```

3. Prior Probability Selection

Prior probabilities reasoning:

General Principles for Choosing Priors (as per "Statistical Rethinking" and good Bayesian practice):

- Priors should reflect knowledge *before* observing the current dataset. They should not be directly determined by summary statistics of the current data, as this amounts to using the data twice.
- Ideally, priors come from previous research, domain expertise, or theoretical constraints.
- When specific prior information is lacking, *weakly informative priors* are used. These are priors that are broad enough to let the data speak, but specific enough to keep the MCMC sampler in a reasonable parameter space and act as a form of regularization.
- **Prior predictive checks** are essential to understand the implications of your chosen priors.

We select the following priors for our model:

Intercept ($\alpha \sim \mathcal{N}(0.5, 0.2)$): Represents the baseline energy consumption for `c-sharp-razor`. The prior is centered at 0.5 Joules, reflecting an assumption based of energy consumption values typically being small but positive. The standard deviation of 0.2 allows for moderate uncertainty, covering a range of plausible baseline values. This choice ensures flexibility while constraining the prior to realistic energy levels.

Framework Effects ($\beta \sim \mathcal{N}(0, 0.2)$): Captures differences in energy consumption between frameworks, and we assume small deviations from the baseline. The prior is centered at 0, as we assume frameworks may not differ significantly in energy consumption. The standard deviation of 0.2 allows for moderate variability, ensuring that the prior is weakly informative and does not overly constrain the model.

Residual SD ($\sigma \sim \text{HalfNormal}(0.25)$): Models unexplained variability in energy consumption, constrained to positive values. The HalfNormal distribution with a scale of 0.25 reflects the expectation that residual variability is relatively small but allows for

sligthly larger values if supported by the data. This choice ensures the model can account for noise while maintaining realistic bounds on the variability. We specifically choose a HalfNormal distribution over an Exponential distribution because the HalfNormal is symmetric around zero (before truncation) and allows for a more gradual decay in probability for larger values, which better aligns with the expectation of moderate variability rather than extreme outliers.

As such, we can specify our **linear model** as:

$$
\begin{aligned}
h_i &\sim \mathcal{N}(\mu_i, \sigma) && [\text{likelihood for observed outcome } h_i] \\
\mu_i &= \alpha + \beta_{F[i]} && [\text{linear model } x \text{ predictor}] \\
\alpha &\sim \mathcal{N}(0.5, 0.2) && [\alpha \text{ prior, parameter}] \\
\beta &\sim \mathcal{N}(0, 0.2) && [\beta \text{ prior, parameter}] \\
\sigma &\sim \text{HalfNormal}(0.25) && [\sigma \text{ prior, parameter}]
\end{aligned}
$$

4. Prior Predictive Checks

We perform prior predictive checks by sampling data from the priors and plotting the resulting distributions of:

1. Simulated energy consumption values (simulated y)
2. Simulated energy means (simulated μ)
3. Distributions of each prior parameter (α, β, and σ)

Additionally, we check the amount of negative simulated energy consumption values and energy means, as we would expect these to be positive.

Performing these prior predictive checks allows us to visualize the implications of our prior choices and ensure they are reasonable before fitting the model to the actual data.

In [8]:
```
# Define the model for prior predictive checks
with pm.Model() as model_h1_prior_pred:
    # Priors
    alpha = pm.Normal('alpha', mu = .5, sigma = .2)
# Baseline energy consumption for c-sharp-razor
    betas = pm.Normal('betas', mu = 0, sigma = .2, shape =
X_h1.shape[1]) # Effects of other frameworks
    sigma = pm.HalfNormal('sigma', sigma = .25)
# Residual standard deviation

    # Deterministic mu for inspection
    mu_deterministic = pm.Deterministic('mu_values', alpha +
pm.math.dot(X_h1.values, betas))
```

```
    # Likelihood for prior predictive sampling
    y_simulated = pm.Normal('y_simulated', mu = mu_deterministic,
sigma = sigma, shape = y.shape[0])

    # Sample from the prior predictive distribution
    prior_pred_samples_h1 = pm.sample_prior_predictive(samples =
500, random_seed = 42)


# Plot the prior predictive distribution for simulated energy
consumption (y_simulated)
print('Plotting Prior Predictive Distribution for y_simulated
(Energy Consumption):')
simulated_y =
prior_pred_samples_h1.prior['y_simulated'].stack(samples =
('chain', 'draw')).values # Stack simulated values
az.plot_dist(simulated_y, kind = 'hist', hist_kwargs = {'alpha':
.8, 'bins': 100}) # Plot using ArviZ
plt.xlabel('Simulated Energy Consumption (Joules)')
plt.ylabel('Density')
plt.title('Prior Predictive Check: Distribution of Simulated
Energy Consumption (y_simulated)')
x_llimit, x_ulimit = np.min(simulated_y), np.max(simulated_y)
plt.xlim(x_llimit, x_ulimit) # Set x-axis limits to 5% and 95%
quantiles
plt.xticks(np.linspace(x_llimit, x_ulimit, num = 15), rotation =
45)
plt.tight_layout()
plt.show()

# Plotting the prior predictive distribution for mu_values
print('\nPlotting Prior Predictive Distribution for mu_values
(Mean Energy Consumption):')
mu_values =
prior_pred_samples_h1.prior['mu_values'].stack(samples =
('chain', 'draw'))
az.plot_dist(mu_values, kind = 'hist', hist_kwargs = {'alpha':
.8, 'bins': 50})
plt.xlabel('Simulated Mean Energy Consumption (Joules)')
plt.ylabel('Density')
plt.title('Prior Predictive Check: Distribution of Simulated Mean
```

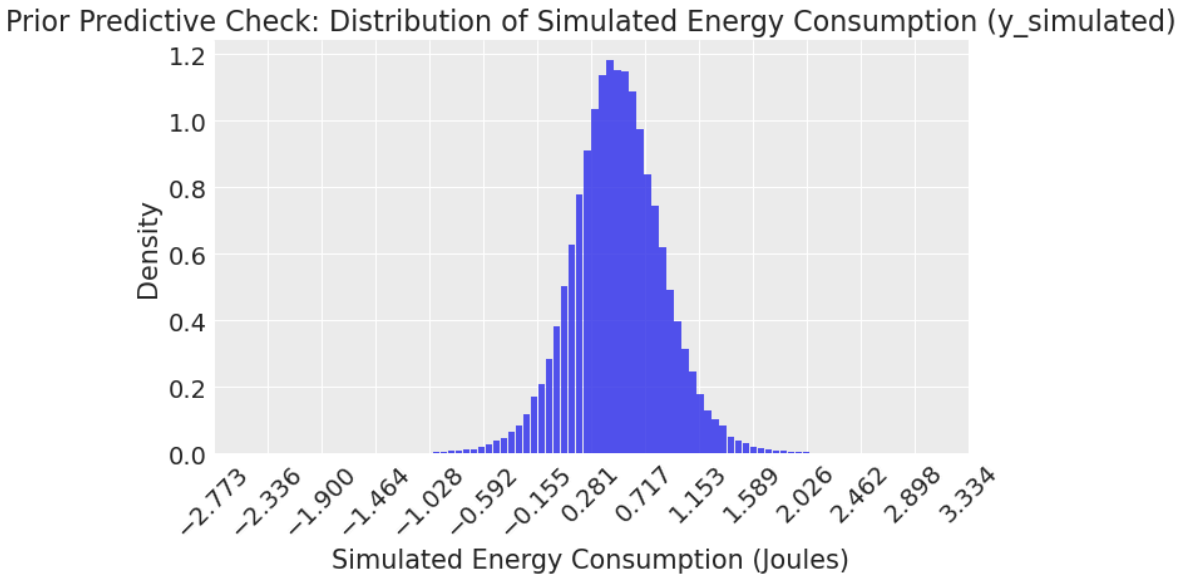
```
Energy (mu_values)')
x_min, x_max = np.min(mu_values), np.max(mu_values)
x_ticks = np.linspace(x_min, x_max, num = 15)
plt.xticks(x_ticks, rotation = 45)
plt.tight_layout()
plt.show()

# Plotting the distributions of the priors themselves for alpha,
betas, and sigma
print('\nPlotting Prior Distributions for Parameters:')
fig, axes = plt.subplots(1, 3, figsize = (18, 5))

# Alphas
alpha_values =
prior_pred_samples_h1.prior['alpha'].stack(samples=('chain',
'draw'))
az.plot_dist(alpha_values, ax = axes[0], label = 'alpha', kind =
'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks = np.linspace(np.min(alpha_values), np.max(alpha_values),
num = 15)
axes[0].set_xticks(x_ticks)
axes[0].set_xticklabels(np.round(x_ticks, 2), rotation = 45)
axes[0].set_title('Prior Distribution for alpha (Baseline
Energy)')
axes[0].set_xlabel('Value')
axes[0].set_ylabel('Density')

# Betas
beta_values = prior_pred_samples_h1.prior['betas'].stack(samples
= ('chain', 'draw'))
az.plot_dist(beta_values, ax = axes[1], label = 'betas', kind =
'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks = np.linspace(np.min(beta_values), np.max(beta_values),
num = 15)
axes[1].set_xticks(x_ticks)
axes[1].set_xticklabels(np.round(x_ticks, 2), rotation = 45)
axes[1].set_title('Prior Distribution for betas (Framework
Effects)')
axes[1].set_xlabel('Value')
axes[1].set_ylabel('Density')

# Sigmas
```

```
sigma_values = prior_pred_samples_h1.prior['sigma'].stack(samples
= ('chain', 'draw'))
az.plot_dist(sigma_values, ax = axes[2], label = 'sigma', kind =
'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks = np.linspace(np.min(sigma_values), np.max(sigma_values),
num = 15)
axes[2].set_xticks(x_ticks)
axes[2].set_xticklabels(np.round(x_ticks, 2), rotation = 45)
axes[2].set_title('Prior Distribution for sigma (Residual SD)')
axes[2].set_xlabel('Value')
axes[2].set_ylabel('Density')

plt.tight_layout()
plt.show()

# Check for negative energy values
simulated_y_flat =
prior_pred_samples_h1.prior['y_simulated'].stack(samples =
('chain', 'draw')).values
negative_y_percentage = np.mean(simulated_y_flat < 0) * 100
print(f'\nPercentage of simulated energy consumption values
(y_simulated) < 0: {negative_y_percentage:.2f}%')

simulated_mu_flat =
prior_pred_samples_h1.prior['mu_values'].stack(samples =
('chain', 'draw')).values
negative_mu_percentage = np.mean(simulated_mu_flat < 0) * 100
print(f'Percentage of simulated mean energy values (mu_values) <
0: {negative_mu_percentage:.2f}%')
```

```
Sampling: [alpha, betas, sigma, y_simulated]
```

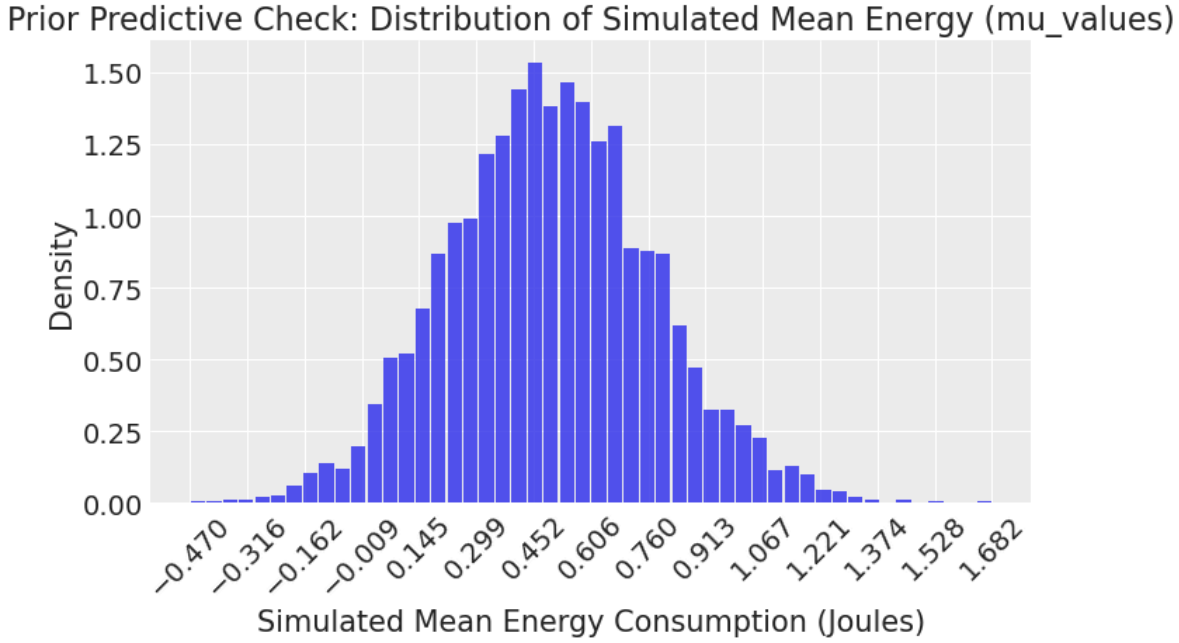
```
Plotting Prior Predictive Distribution for y_simulated (Energy Consumptio
n):
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/179343481
9.py:28: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



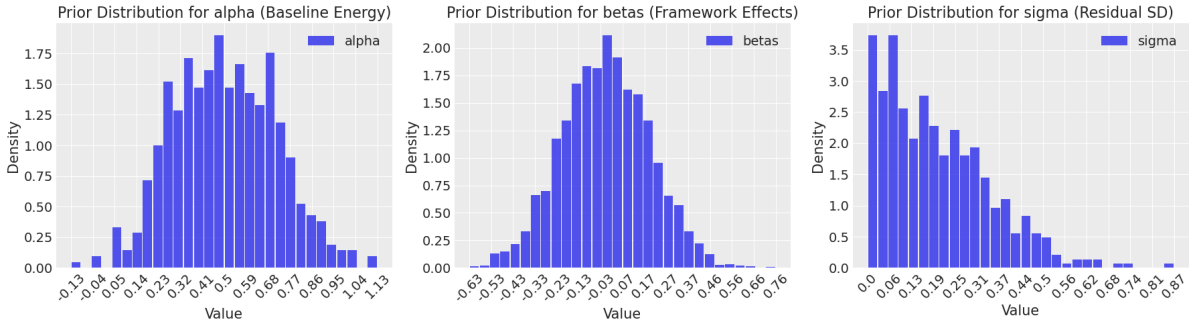

```
Plotting Prior Predictive Distribution for mu_values (Mean Energy Consumpti
on):
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/179343481
9.py:41: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```




```
Plotting Prior Distributions for Parameters:
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/179343481
9.py:78: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```




```
Percentage of simulated energy consumption values (y_simulated) < 0: 7.65%
Percentage of simulated mean energy values (mu_values) < 0: 2.91%
```

Interpretation of Prior Predictive Checks for H1

The plots generated by the prior predictive checks in the cell above help us to better understand the implications of the chosen priors:

1. **Distribution of Simulated Energy Consumption (`y_simulated`)**: The simulated energy consumption values align well with plausible energy levels. The range of simulated values is consistent with what we might intuitively expect for energy consumption in this context. The proportion of negative values is fairly small at 7.65%, indicating that the priors for α and σ constrain the simulated values to realistic ranges.

2. **Distribution of Simulated Mean Energy Consumption (`mu_values`)**: The simulated mean energy consumption values are predominantly positive, with only 2.91% negative values. This demonstrates that the prior for the mean energy consumption is at least modestly well-calibrated, ensuring that the simulated values are centered around plausible energy levels.

3. **Prior Distributions for Parameters (α, β, σ)**: The prior distribution plots seem to follow the specified prior distribution parameters well.

5. Model Fitting and Sampling

We fit the model with `PyMC` and sample from the posterior distribution using the No-U-Turn Sampler (NUTS). The model is specified in a `with` block, and we use `pm.sample()` to draw samples from the posterior distribution. We set the number of samples to 2000 and the number of tuning steps to 1000. We use `chains=4` to run four chains in parallel for better convergence diagnostics.

Furthermore, a visualization of the model setup is printed through `pm.model_to_graphviz`.

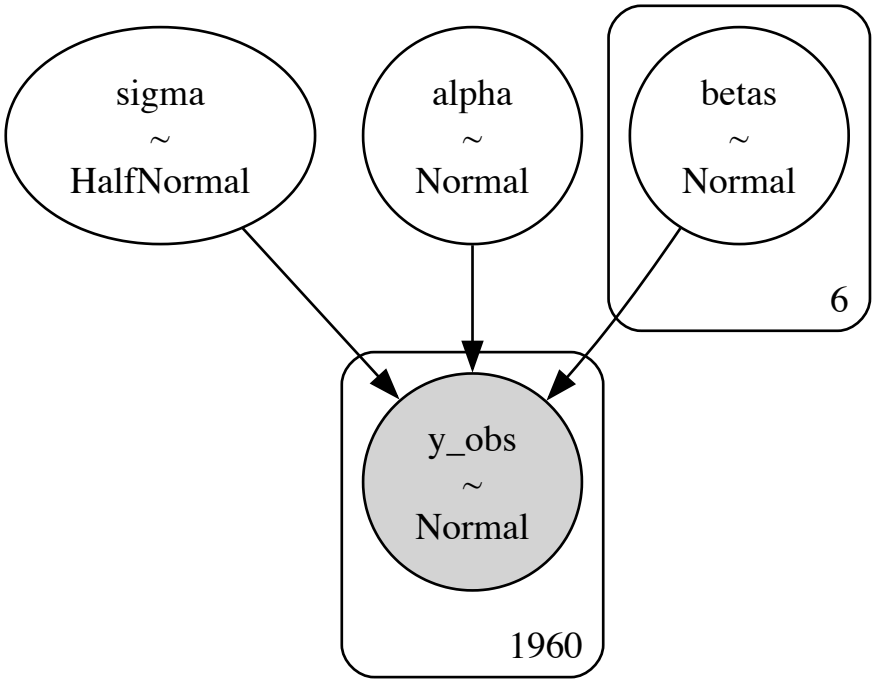
In [9]:
```
# Define the Bayesian model
with pm.Model() as model_h1:
    # Priors
    alpha = pm.Normal('alpha', mu = .5, sigma = .2)
# Baseline energy consumption for c-sharp-razor
    betas = pm.Normal('betas', mu = 0, sigma = .2, shape =
X_h1.shape[1]) # Effects of other frameworks
    sigma = pm.HalfNormal('sigma', sigma = .25)
# Residual standard deviation

    # Define linear model
    mu = alpha + pm.math.dot(X_h1.values, betas)
```

```
    # Likelihood
    y_obs = pm.Normal('y_obs', mu = mu, sigma = sigma, observed =
y)

    # Sampling
    trace_h1 = pm.sample(2000, tune = 1000, target_accept = .95,
return_inferencedata = True, random_seed = 42, idata_kwargs=
{'log_likelihood': True})

pm.model_to_graphviz(model_h1)
```

```
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [alpha, betas, sigma]
Output()
```

```
Sampling 4 chains for 1_000 tune and 2_000 draw iterations (4_000 + 8_000 d
raws total) took 18 seconds.
```

Out[9]:



6. Trace Plot and Summary Analysis

We use **trace plots** to visualize the posterior distributions generated by the four chains in the sampling process. We use **summary statistics** to assess the convergence, quality of estimates, and statistical significance.

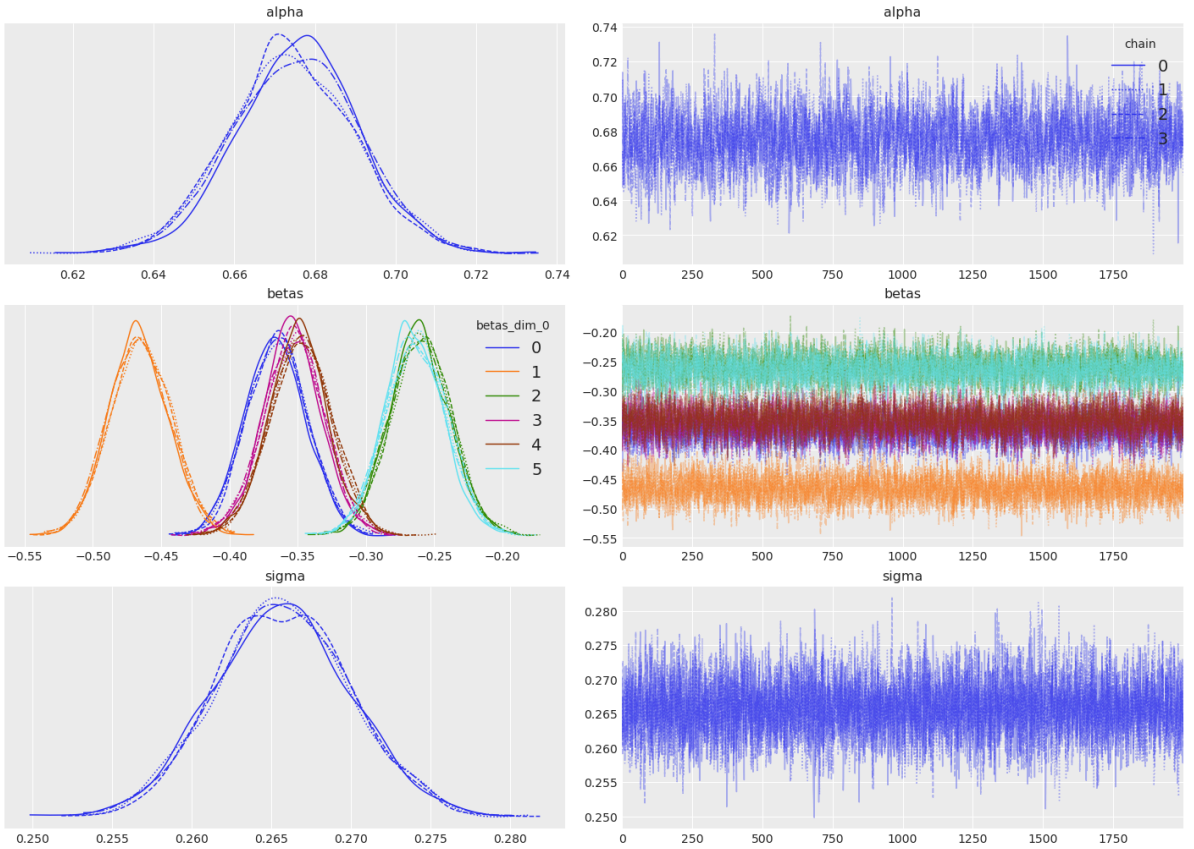
In [10]:

```
# Posterior analysis
az.plot_trace(trace_h1, var_names = ['alpha', 'betas', 'sigma'],
figsize = (14, 10), legend = True)
```

```
az.summary(trace_h1, var_names = ['alpha', 'betas', 'sigma'],
round_to = 4)
```

Out[10]:

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail
alpha	0.6748	0.0155	0.6460	0.7044	0.0003	0.0002	2336.5269	3107.3791
betas[0]	-0.3655	0.0221	-0.4068	-0.3239	0.0004	0.0003	3216.0903	4559.9633
betas[1]	-0.4656	0.0222	-0.5075	-0.4240	0.0004	0.0003	3196.9790	3818.6724
betas[2]	-0.2601	0.0221	-0.3006	-0.2181	0.0004	0.0003	3245.9831	4772.5086
betas[3]	-0.3533	0.0220	-0.3950	-0.3111	0.0004	0.0003	3316.7905	4453.7754
betas[4]	-0.3482	0.0223	-0.3893	-0.3053	0.0004	0.0003	3229.3326	4938.9423
betas[5]	-0.2638	0.0221	-0.3048	-0.2225	0.0004	0.0003	3275.1523	4984.9502
sigma	0.2657	0.0043	0.2575	0.2736	0.0001	0.0000	6925.6367	5383.3112



Interpretation of Trace Plots and Summary Statistics for H1

1. **Trace Plots**

- Trace plots for α and σ are generally stationary and without trends, converging around a consistent range of values.
- Trace plots for the beta coefficients imply that the frameworks can be categorized into three groups based on their energy consumption relative to the baseline:
  1. β_1 (lowest on the y-axis, centered around approx. -0.47): The β_1 trace plot has the lowest energy consumption when compared to the baseline.
  2. β_0, β_3, β_4 (centered around approx. -0.35): The three frameworks in this group also seem to consume less energy than the baseline, but more than β_1. Since

these beta coefficients overlap, the differences in energy consumption seem to be insignificant.

3. β_2, β_5 (centered around approx. -0.27): The frameworks in this group consume the most energy among the non-baseline frameworks, but still less than the baseline. Since these beta coefficients overlap, the difference in energy consumption between them seems to be insignificant.

1. **Posterior Distrubutions**

- The distributions are unimodal and do not look overly wide or flat

1. **Summary Table**

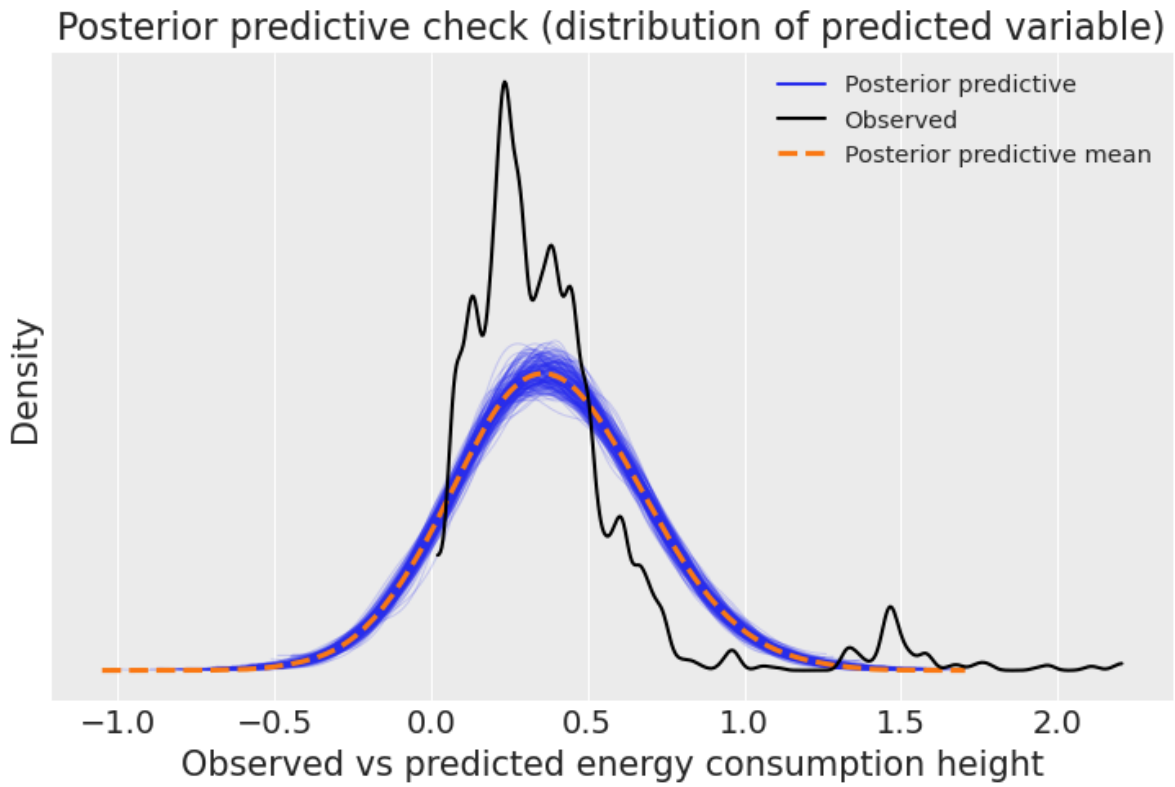
- `R-hat` is close to 1 for all parameters, indicating good convergence.
- `ESS` are sufficiently large for all parameters, indicating reliable estimates.
- `HDI` does not include 0 for any betas, indicating that the effects are statistically significant.

7. Posterior Predictive Checks

We perform posterior predictive checks to assess the model's fit to the data. By sampling from the posterior distribution, we can compare the observed data to the predicted responses. We plot the observed energy consumption values against the predicted values.

In [11]:
```
# Generate posterior predictive samples
with model_h1:
    ppc_h1 = pm.sample_posterior_predictive(trace_h1, var_names =
["y_obs"], random_seed = 42)

_, ax = plt.subplots()
az.plot_ppc(ppc_h1, num_pp_samples = 200, ax = ax)
ax.set_xlabel('Observed vs predicted energy consumption height')
ax.set_ylabel('Density')
ax.set_title('Posterior predictive check (distribution of
predicted variable)');
```

```
Sampling: [y_obs]
Output()
```



Interpretation of Posterior Predictive Checks for H1

The plot shows the observed energy consumption values against the predictions from the posterior predictive checks. We see that the predictions fit the observed data fairly well, although the distribution seems to have a slightly higher standard deviation. Furthermore, the predictions still produce some negative values.

However, given the checks made from trace plots and summary statistics, as well as the amount of noise in the observed data, we still believe the model fit is acceptable.

8. Hypothesis Conclusion

While the trace plots indicate that all other frameworks consume less energy than the baseline, we still intent to test H1 more statistically.

To do so, we quantify the evidence for each framework by computing the proportion of posterior samples where $\beta_j < 0$. This gives you the posterior probability $P(\beta_j < 0|\text{data})$

In [12]:
```
# Posterior_betas will be a 3D array: (chains, draws,
num_other_frameworks)
posterior_betas = trace_h1.posterior['betas']

# Calculate probability for each beta
prob_beta_negative = (posterior_betas < 0).mean(dim = ('chain',
'draw'))
```

```
# To see these probabilities with framework names (assuming
X.columns has the names)
framework_names = X_h1.columns # From your data prep cell
for i, name in enumerate(framework_names):
    print(f"P(beta_{name} < 0) =
{prob_beta_negative[i].item():.3f}")
```

```
P(beta_actix < 0) = 1.000
P(beta_express < 0) = 1.000
P(beta_flask < 0) = 1.000
P(beta_gin < 0) = 1.000
P(beta_gorilla < 0) = 1.000
P(beta_sinatra < 0) = 1.000
```

As such, we can conclude that the model provides strong evidence supporting H1, i.e., that the web framework `c-sharp-razor` consumes more energy than any other web framework in the dataset. The posterior probabilities indicate that all non-baseline frameworks are virtually certain ($P(\beta < 0) = 1$) to consume less energy than `c-sharp-razor`.

The trace plots and summary statistics further support this: all chains mix well, `R-hat` values are close to 1, and effective sample sizes are high, indicating good convergence and reliable estimates. Posterior predictive checks demonstrate that the model captures the observed data distribution reasonably well, however with some deviations. Taken together, these results provide compelling evidence in support of H1.

Hypothesis H2

* **H2** - The programming language `javascript` consumes the least amount of energy compared to any other programming language in the dataset.

To evaluate hypothesis **H2** we design a Bayesian regression model, predicting energy consumption based on programming language. The implication of H2 is very similar to H1, but we are now interested in the effects of programming language rather than web framework.

As such, we can assume a very similar **linear model**, specified as follows:

$$
\begin{aligned}
h_i &\sim \mathcal{N}(\mu_i, \sigma) && [\text{likelihood for observed outcome } h_i] \\
\mu_i &= \alpha + \beta_{L[i]} && [\text{linear model } x \text{ predictor}]
\end{aligned}
$$

The likelihood assumes that the observed energy consumption values h_i are normally distributed around the predicted mean μ_i with a constant standard deviation σ.

The mean μ_i is modeled as a linear combination of the intercept α and the language effects β.

The prior probailities for parameters α, β, and σ are chosen based on intution and domain knowledge, as explained later.

The modeling process again follows the below steps:

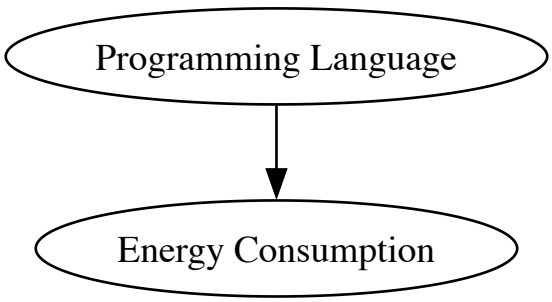
1. Model visualization
2. Data preparation
3. Prior probability selection
4. Prior predictive checks
5. Model fitting and sampling
6. Trace plot and summary analysis
7. Posterior predictive checks
8. Hypothesis conclusion

1. Model Visualization

First, we draw the model design as a simple directed acyclic graph (DAG).

In [13]:
```
# Initializing a Causal Inference Model (DAG: Directed Acyclic Graph)
CausalGraphicalModel(
    nodes=["Programming Language", "Energy Consumption"],
    edges=[
        ("Programming Language", "Energy Consumption"),
    ]
).draw()
```

Out[13]:



2. Data Preparation

Creating dummy variables for the categorical variable `language` allows us to use them as individual predictors in the regression model. Since we hypothesize that `javascript` consumes the least energy, we use this language as the *baseline* category. As such, we exclude its dummy variable from the regression model. This means the coefficients of other languages will be interpreted relative to `javascript`.

In [14]:
```
# Make javascript the baseline language
df['language'] = df['language'].astype('category')
df['language'] = df['language'].cat.reorder_categories(
    ['javascript'] + [cat for cat in
df['language'].cat.categories if cat != 'javascript'],
    ordered = False
)

# Combine with original data
X_h2 = pd.get_dummies(df['language'], drop_first = True)
y = df['energy_consumption'].values
```

3. Prior Probability Selection

Given the similarity of the model to H1, we can use the same prior probabilities for the parameters.

The only difference is that we now use the language as the predictor instead of the framework, essentially changing the interpretation of the β coefficients. However, we would still have a base assumption that the change in energy consumption based on the programming language would be 0, and thus we set the prior for β to be centered around 0. We similarly keep the standard deviation of 0.2 for β to allow for moderate variability.

As such, we can specify our **linear model** as:

$$
\begin{aligned}
h_i &\sim \mathcal{N}(\mu_i, \sigma) && [\text{likelihood for observed outcome } h_i] \\
\mu_i &= \alpha + \beta_{L[i]} && [\text{linear model } x \text{ predictor}] \\
\alpha &\sim \mathcal{N}(0.5, 0.2) && [\alpha \text{ prior, parameter}] \\
\beta &\sim \mathcal{N}(0, 0.2) && [\beta \text{ prior, parameter}] \\
\sigma &\sim \text{HalfNormal}(0.25) && [\sigma \text{ prior, parameter}]
\end{aligned}
$$

4. Prior Predictive Checks

We perform prior predictive checks by sampling data from the priors and plotting the resulting distributions of:

1. Simulated energy consumption values (simulated y)
2. Simulated energy means (simulated μ)
3. Distributions of each prior parameter (α, β, and σ)

Additionally, we check the amount of negative simulated energy consumption values and energy means, as we would expect these to be positive.

Performing these prior predictive checks allows us to visualize the implications of our prior choices and ensure they are reasonable before fitting the model to the actual data.

In [15]:

```
# Define the model for prior predictive checks
with pm.Model() as model_h2_prior_pred:
    # Priors
    alpha = pm.Normal('alpha', mu =  .5, sigma = .2)
# Baseline energy consumption for javascript
    betas = pm.Normal('betas', mu =  0, sigma = .2, shape =
X_h2.shape[1]) # Effects of other languages
    sigma = pm.HalfNormal('sigma', sigma = .25)
# Residual standard deviation

    # Deterministic mu for inspection
    mu_deterministic = pm.Deterministic('mu_values', alpha +
pm.math.dot(X_h2.values, betas))

    # Likelihood for prior predictive sampling
    y_simulated = pm.Normal('y_simulated', mu = mu_deterministic,
sigma = sigma, shape = y.shape[0])

    # Sample from the prior predictive distribution
    prior_pred_samples_h2 = pm.sample_prior_predictive(samples =
500, random_seed = 42)

# Plot the prior predictive distribution for simulated energy
consumption (y_simulated)
print('Plotting Prior Predictive Distribution for y_simulated
(Energy Consumption):')
simulated_y =
prior_pred_samples_h2.prior['y_simulated'].stack(samples =
('chain', 'draw')).values  # Stack simulated values
```

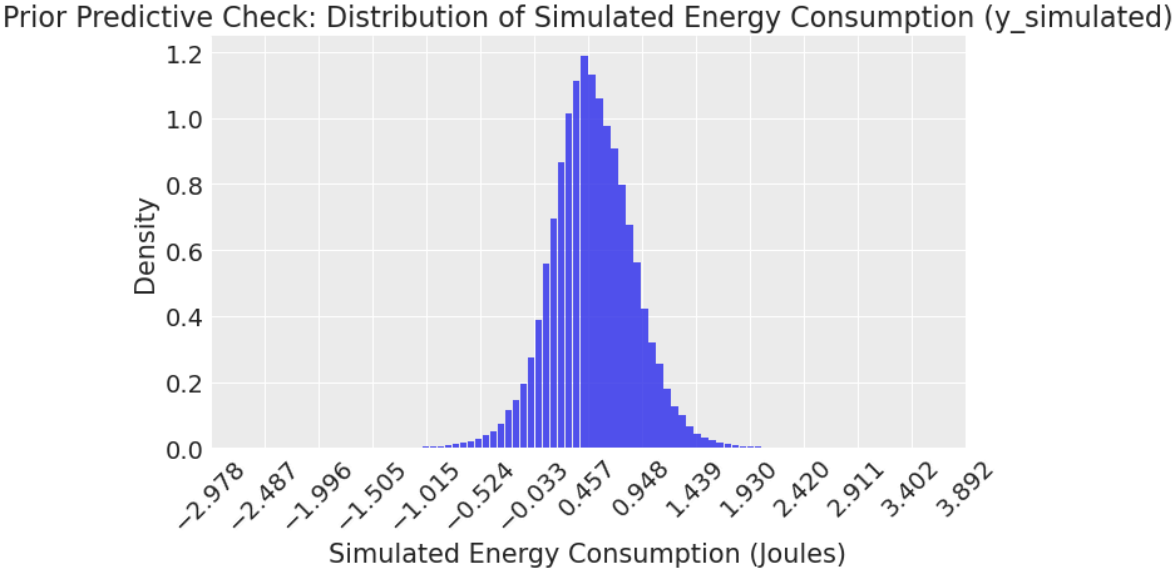
```
az.plot_dist(simulated_y, kind = 'hist', hist_kwargs = {'alpha':
.8, 'bins': 100})  # Plot using ArviZ
plt.xlabel('Simulated Energy Consumption (Joules)')
plt.ylabel('Density')
plt.title('Prior Predictive Check: Distribution of Simulated
Energy Consumption (y_simulated)')
x_llimit, x_ulimit = np.min(simulated_y), np.max(simulated_y)
plt.xlim(x_llimit, x_ulimit)  # Set x-axis limits
plt.xticks(np.linspace(x_llimit, x_ulimit, num = 15), rotation =
45)
plt.tight_layout()
plt.show()

# Plotting the prior predictive distribution for mu_values
print('\nPlotting Prior Predictive Distribution for mu_values
(Mean Energy Consumption):')
mu_values =
prior_pred_samples_h2.prior['mu_values'].stack(samples =
('chain', 'draw'))
az.plot_dist(mu_values, kind = 'hist', hist_kwargs = {'alpha':
.8, 'bins': 50})
plt.xlabel('Simulated Mean Energy Consumption (Joules)')
plt.ylabel('Density')
plt.title('Prior Predictive Check: Distribution of Simulated Mean
Energy (mu_values)')
x_min, x_max = np.min(mu_values), np.max(mu_values)
x_ticks = np.linspace(x_min, x_max, num = 15)
plt.xticks(x_ticks, rotation = 45)
plt.tight_layout()
plt.show()

# Plotting the distributions of the priors themselves for alpha,
betas, and sigma
print('\nPlotting Prior Distributions for Parameters:')
fig, axes = plt.subplots(1, 3, figsize = (18, 5))

# Alphas
alpha_values = prior_pred_samples_h2.prior['alpha'].stack(samples
= ('chain', 'draw'))
az.plot_dist(alpha_values, ax = axes[0], label = 'alpha', kind =
'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks = np.linspace(np.min(alpha_values), np.max(alpha_values),
```

```
num = 15)
axes[0].set_xticks(x_ticks)
axes[0].set_xticklabels(np.round(x_ticks, 2), rotation = 45)
axes[0].set_title('Prior Distribution for alpha (Baseline
Energy)')
axes[0].set_xlabel('Value')
axes[0].set_ylabel('Density')

# Betas
beta_values = prior_pred_samples_h2.prior['betas'].stack(samples
= ('chain', 'draw'))
az.plot_dist(beta_values, ax = axes[1], label = 'betas', kind =
'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks = np.linspace(np.min(beta_values), np.max(beta_values),
num = 15)
axes[1].set_xticks(x_ticks)
axes[1].set_xticklabels(np.round(x_ticks, 2), rotation = 45)
axes[1].set_title('Prior Distribution for betas (Language
Effects)')
axes[1].set_xlabel('Value')
axes[1].set_ylabel('Density')

# Sigmas
sigma_values = prior_pred_samples_h2.prior['sigma'].stack(samples
= ('chain', 'draw'))
az.plot_dist(sigma_values, ax = axes[2], label = 'sigma', kind =
'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks = np.linspace(np.min(sigma_values), np.max(sigma_values),
num = 15)
axes[2].set_xticks(x_ticks)
axes[2].set_xticklabels(np.round(x_ticks, 2), rotation = 45)
axes[2].set_title('Prior Distribution for sigma (Residual SD)')
axes[2].set_xlabel('Value')
axes[2].set_ylabel('Density')

plt.tight_layout()
plt.show()

# Check for negative energy values
simulated_y_flat =
prior_pred_samples_h2.prior['y_simulated'].stack(samples =
('chain', 'draw')).values
```

```
negative_y_percentage = np.mean(simulated_y_flat < 0) * 100
print(f'\nPercentage of simulated energy consumption values
(y_simulated) < 0: {negative_y_percentage:.2f}%')

simulated_mu_flat =
prior_pred_samples_h2.prior['mu_values'].stack(samples =
('chain', 'draw')).values
negative_mu_percentage = np.mean(simulated_mu_flat < 0) * 100
print(f'Percentage of simulated mean energy values (mu_values) <
0: {negative_mu_percentage:.2f}%')
```

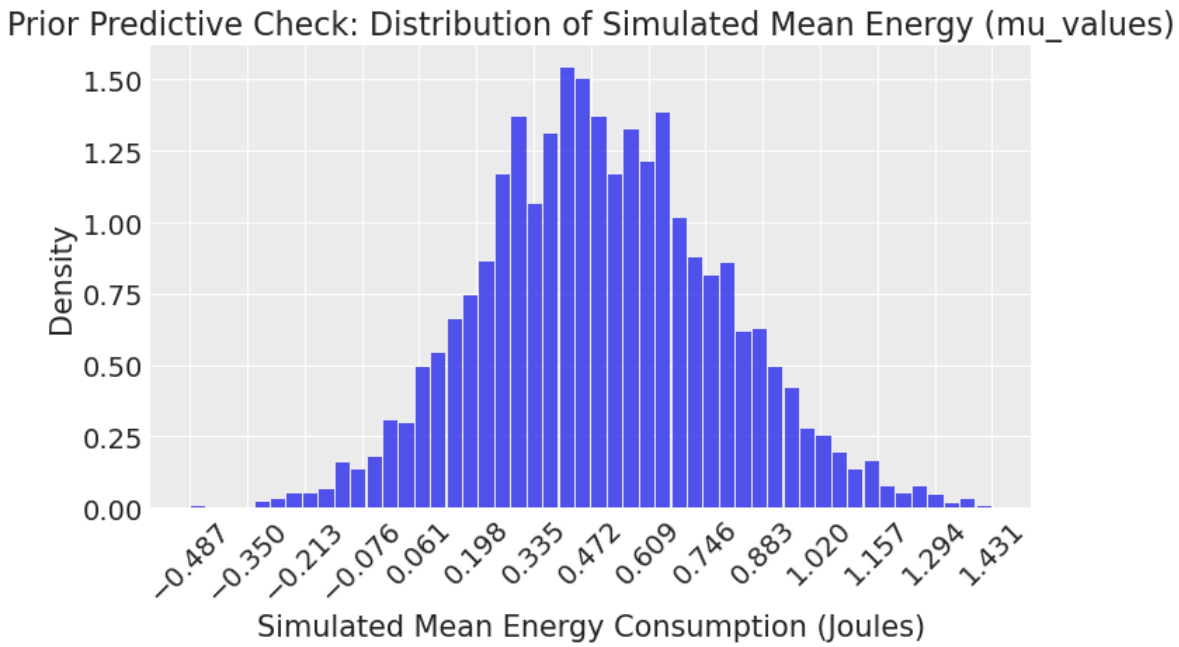
```
Sampling: [alpha, betas, sigma, y_simulated]
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/220116730
4.py:27: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```

```
Plotting Prior Predictive Distribution for y_simulated (Energy Consumptio
n):
```



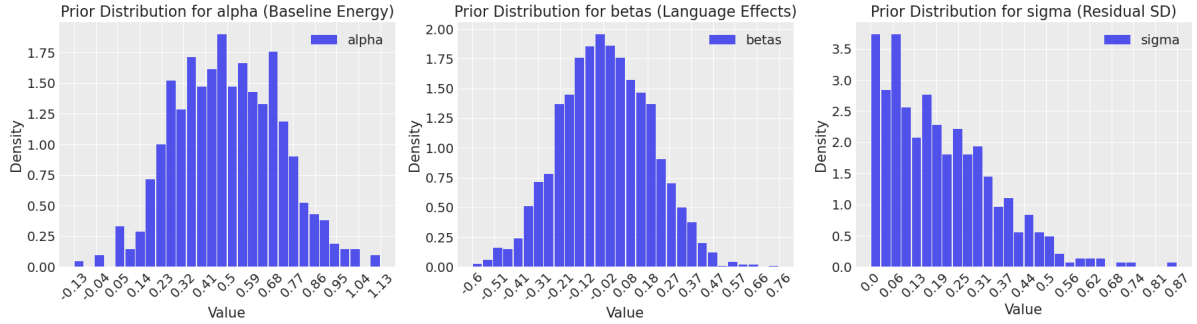

```
Plotting Prior Predictive Distribution for mu_values (Mean Energy Consumpti
on):
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/220116730
4.py:40: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```




```
Plotting Prior Distributions for Parameters:
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/2201167304.py:77: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```




```
Percentage of simulated energy consumption values (y_simulated) < 0: 7.60%
Percentage of simulated mean energy values (mu_values) < 0: 3.51%
```

Interpretation of Prior Predictive Checks for H2

The plots generated by the prior predictive checks in the cell above help us to better understand the implications of the chosen priors:

1. **Distribution of Simulated Energy Consumption (`y_simulated`)**: The simulated energy consumption again values align well with plausible energy levels. The range of simulated values is consistent with what we might intuitively expect for energy consumption in this context, and are also extremely close to the simulated energy consumption values from H1. This makes intuitive sense given that the prior parameters have not changed. The proportion of negative values is thus fairly small at 7.6%, indicating that the priors for α and σ constrain the simulated values to realistic ranges.

2. **Distribution of Simulated Mean Energy Consumption (`mu_values`)**: The simulated mean energy consumption values are again mostly positive, with only 3.51% negative values. This demonstrates that the prior for the mean energy consumption is still modestly well-calibrated and that the simulated values are centered around plausible energy levels.

3. **Prior Distributions for Parameters (α, β, σ)**: The prior distribution plots seem to follow the specified prior distribution parameters well.

5. Model Fitting and Sampling

We fit the model with `PyMC` and sample from the posterior distribution using the No-U-Turn Sampler (NUTS). The model is specified in a `with` block, and we use `pm.sample()` to draw samples from the posterior distribution. We set the number of samples to 2000 and the number of tuning steps to 1000. We use `chains=4` to run four chains in parallel for better convergence diagnostics.

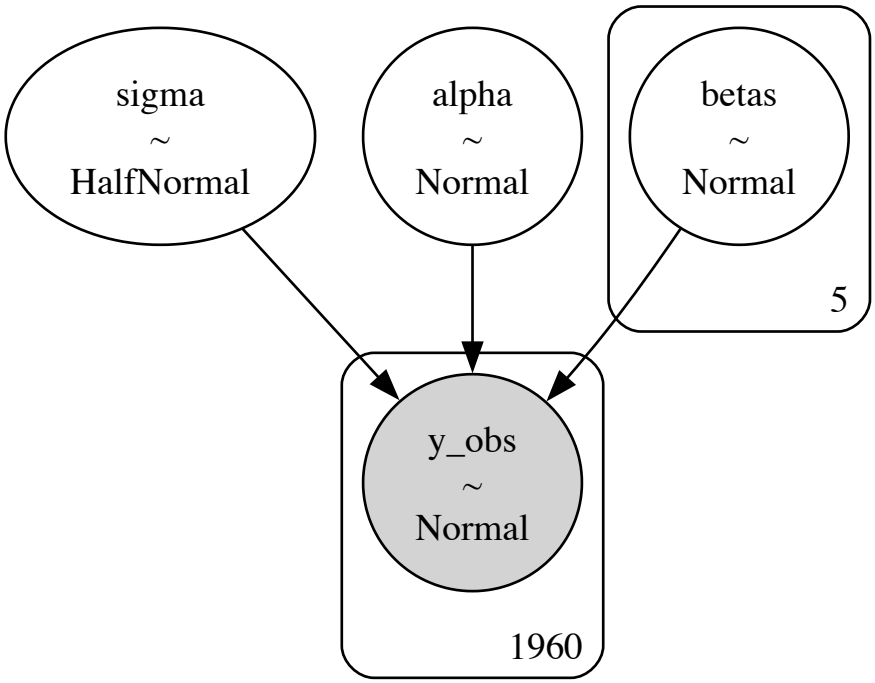
Furthermore, a visualization of the model setup is printed through `pm.model_to_graphviz`.

In [16]:
```
# Define the Bayesian model for H2
with pm.Model() as model_h2:
    # Priors
    alpha = pm.Normal('alpha', mu =  .5, sigma = .2)
# Baseline energy consumption for javascript
    betas = pm.Normal('betas', mu =  0, sigma = .2, shape =
X_h2.shape[1]) # Effects of other languages
    sigma = pm.HalfNormal('sigma', sigma = .25)
# Residual standard deviation

    # Define the linear model
    mu = alpha + pm.math.dot(X_h2.values, betas)

    # Likelihood
    y_obs = pm.Normal('y_obs', mu = mu, sigma = sigma, observed =
y)

    # Sampling
    trace_h2 = pm.sample(2000, tune = 1000, target_accept = 0.95,
return_inferencedata = True, random_seed = 42, idata_kwargs=
{'log_likelihood': True})

pm.model_to_graphviz(model_h2)
```

```
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [alpha, betas, sigma]
Output()
```

```
Sampling 4 chains for 1_000 tune and 2_000 draw iterations (4_000 + 8_000 d
raws total) took 17 seconds.
```

Out[16]:



6. Trace Plot and Summary Analysis

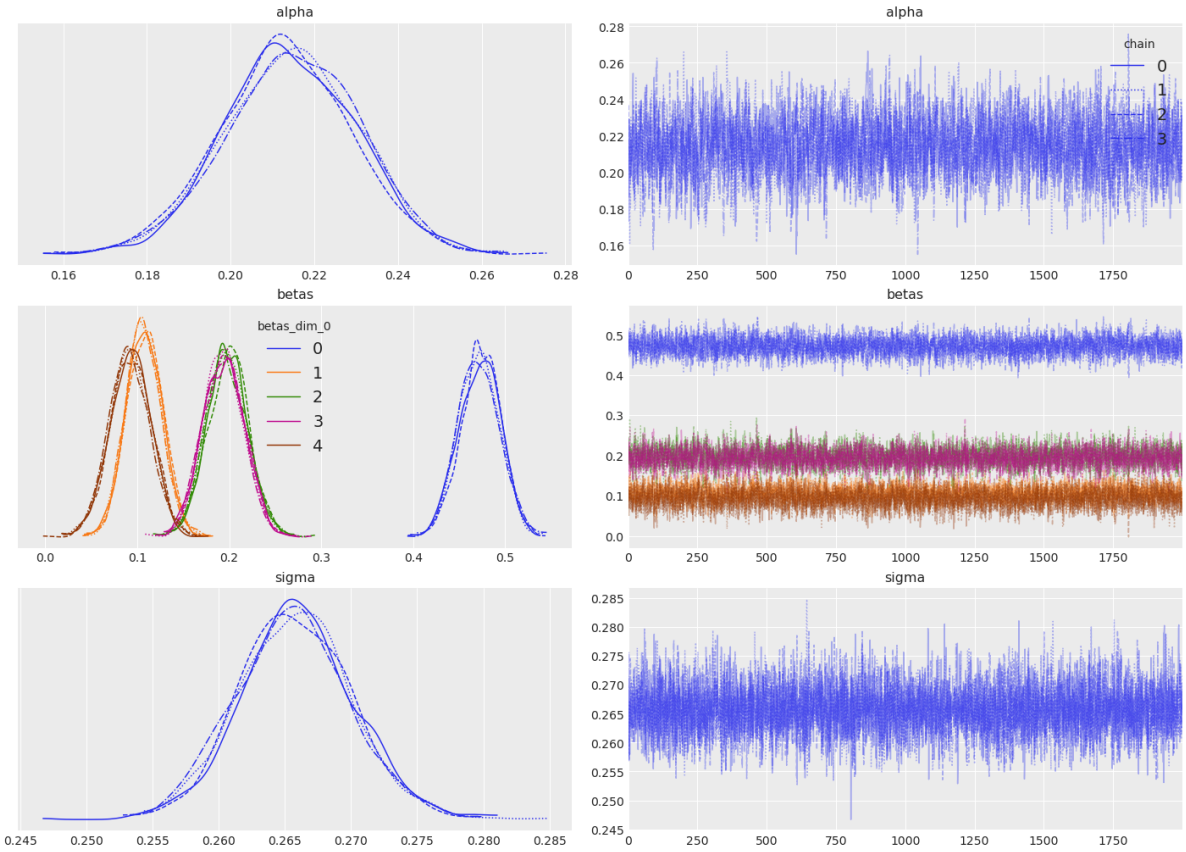
We use **trace plots** to visualize the posterior distributions generated by the four chains in the sampling process. We use **summary statistics** to assess the convergence, quality of estimates, and statistical significance.

In [17]:

```
# Posterior analysis
az.plot_trace(trace_h2, var_names = ['alpha', 'betas', 'sigma'],
figsize = (14, 10), legend = True)
az.summary(trace_h2, var_names = ['alpha', 'betas', 'sigma'],
round_to = 4)
```

Out[17]:

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail
alpha	0.2142	0.0158	0.1828	0.2417	0.0003	0.0002	2372.2638	2805.2380
betas[0]	0.4723	0.0221	0.4309	0.5129	0.0004	0.0003	3273.7242	4249.6233
betas[1]	0.1074	0.0195	0.0712	0.1440	0.0004	0.0003	2991.4682	3966.0372
betas[2]	0.1978	0.0224	0.1558	0.2395	0.0004	0.0003	3336.7148	4424.9881
betas[3]	0.1937	0.0223	0.1526	0.2350	0.0004	0.0003	3227.3530	4885.3837
betas[4]	0.0923	0.0222	0.0504	0.1332	0.0004	0.0003	2863.0893	4397.8580
sigma	0.2658	0.0043	0.2580	0.2742	0.0001	0.0000	6456.3821	5173.3197



Interpretation of Trace Plots and Summary Statistics for H2

1. **Trace Plots**

- Trace plots for α and σ are generally stationary and without trends, converging around a consistent range of values.
- Trace plots for the beta coefficients imply that the languages can be categorized into three groups based on their energy consumption relative to the baseline:
    1. β_0 (highest on the y-axis, centered around approx. 0.47): The β_0 trace plot has the highest energy consumption when compared to the baseline.
    2. β_2, β_3 (centered around approx. 0.2): The two languages in this group also seem to consume more energy than the baseline, but less than the β_0. Since these beta coefficients overlap, the differences in energy consumption between them1seem to be insignificant.
    3. β_1, β_1 (centered around approx. 0.1): The languages in this group consume the least energy among the non-baseline frameworks, but still more than the baseline. Since these beta coefficients overlap, the difference in energy consumption between them seems to be insignificant.

1. **Posterior Distrubutions**

- The distributions are unimodal and do not look overly wide or flat

1. **Summary Table**

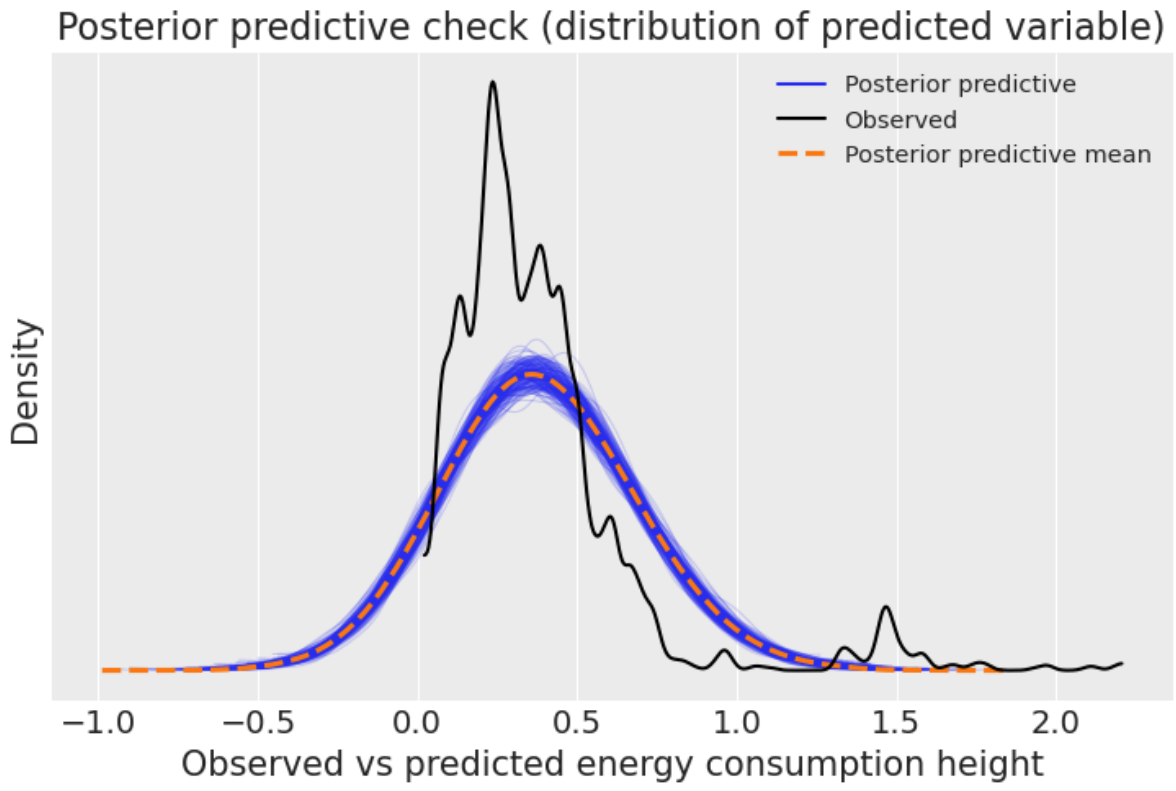
- `R-hat` is close to 1 for all parameters, indicating good convergence.
- `ESS` are sufficiently large for all parameters, indicating reliable estimates.
- `HDI` does not include 0 for any betas, indicating that the effects are statistically significant.

7. Posterior Predictive Checks

We perform posterior predictive checks to assess the model's fit to the data. By sampling from the posterior distribution, we can compare the observed data to the predicted responses. We plot the observed energy consumption values against the predicted values.

In [18]:
```
# Generate posterior predictive samples
with model_h2:
    ppc_h2 = pm.sample_posterior_predictive(trace_h2, var_names =
['y_obs'], random_seed = 42)

_, ax = plt.subplots()
az.plot_ppc(ppc_h2, num_pp_samples = 200, ax = ax)
ax.set_xlabel('Observed vs predicted energy consumption height')
ax.set_ylabel('Density')
ax.set_title('Posterior predictive check (distribution of
predicted variable)');
```

```
Sampling: [y_obs]
Output()
```



Interpretation of Posterior Predictive Checks for H2

The plot shows the observed energy consumption values against the predictions from the posterior predictive checks. We see that the predictions fit the observed data fairly well, although the distribution seems to have a slightly higher standard deviation. Furthermore, the predictions still produce some negative values.

However, given the checks made from trace plots and summary statistics, as well as the amount of noise in the observed data, we still believe the model fit is acceptable.

It should be noted that **this posterior predictive check plot is essentially identical to the one for H1**. This is expected, as the two models for H1 and H2 are very similar, with the only difference being the predictor variable. Furthermore, the two predictor variables (`framework` and `language`) are highly correlated with an almost 1-to-1 mapping. All languages map to exactly one framework each, except for `go` , which uses either `gorilla` or `gin` .

8. Hypothesis Conclusion

While the trace plots indicate that all other frameworks consume more energy than the baseline, we still intent to test H2 more statistically.

To do so, we quantify the evidence for each framework by computing the proportion of posterior samples where $\beta_j > 0$. This gives us the posterior probabilities $P(\beta_j > 0|\text{data})$

In [19]:
```
# Posterior_betas will be a 3D array: (chains, draws,
num_other_languagea)
posterior_betas = trace_h2.posterior['betas']

# Calculate probability for each beta
prob_beta_positive = (posterior_betas > 0).mean(dim = ('chain',
'draw'))

# To see these probabilities with language names (assuming
X.columns has the names)
framework_names = X_h2.columns # From your data prep cell
for i, name in enumerate(framework_names):
    print(f"P(beta_{name} > 0) =
{prob_beta_positive[i].item():.3f}")
```

```
P(beta_c-sharp > 0) = 1.000
P(beta_go > 0) = 1.000
P(beta_python > 0) = 1.000
P(beta_ruby > 0) = 1.000
P(beta_rust > 0) = 1.000
```

As such, we can conclude that the model provides strong evidence supporting H2, i.e., that the programming language `javascript` consumes less energy than any other web framework in the dataset. The posterior probabilities indicate that all non-baseline frameworks are virtually certain ($P(\beta > 0) = 1$) to consume more energy than `javascript`.

The trace plots and summary statistics further support this: all chains mix well, `R–hat` values are close to 1, and effective sample sizes are high, indicating good convergence and reliable estimates. Posterior predictive checks demonstrate that the model captures the observed data distribution reasonably well, however with some deviations. Taken together, these results provide compelling evidence in support of H2.

Hypothesis H3

* **H3** - Runtime has a stronger impact on energy consumption for some API endpoints than others. That is, the effect of runtime on energy consumption is larger for some API endpoints than others.

To evaluate hypothesis **H3** we design a Bayesian regression model, predicting energy consumption based on runtime with **API endpoint-specific intercepts and slopes**. This allows us to model the varying effects of runtime on energy consumption across different API endpoints.

We are assuming a **hierarchical model**, specified as follows:

$$
\begin{aligned}
h_i &\sim \mathcal{N}(\mu_i, \sigma) && [\text{likelihood for observed energy consumption}] \\
\mu_i &= \alpha_{\text{ep}[i]} + \beta_{\text{ep}[i]} \cdot R_i && [\text{multilevel model with runtime predictor}]
\end{aligned}
$$

Where:

- $\alpha_{\text{ep}[i]}$ is the endpoint-specific intercept, representing the baseline energy consumption for each API endpoint.
- $\beta_{\text{ep}[i]}$ is the endpoint-specific slope, representing the effect of runtime on energy consumption for each API endpoint.
- $\text{R}[i]$ is the runtime, which serves as the explanatory predictor variable.
- σ is the residual standard deviation, representing the variability in energy consumption not explained by the model.

This is also referred to as a **varying intercept and varying slope model**, or a **hierarchical model**.

The likelihood assumes that the observed energy consumption values h_i are normally distributed around the predicted mean μ_i with a constant standard deviation σ.

The mean μ_i is modeled as a linear combination of the intercept α and the runtime effect β, both of which vary by API endpoint. This hierarchical structure allows us to partially pool information across endpoints, improving estimates for endpoints with sparse data.

The prior probabilities for parameters α, β, and σ are chosen based on intuition, exploratory data analysis, and domain knowledge, as explained later.

The modeling process then follows the below steps:

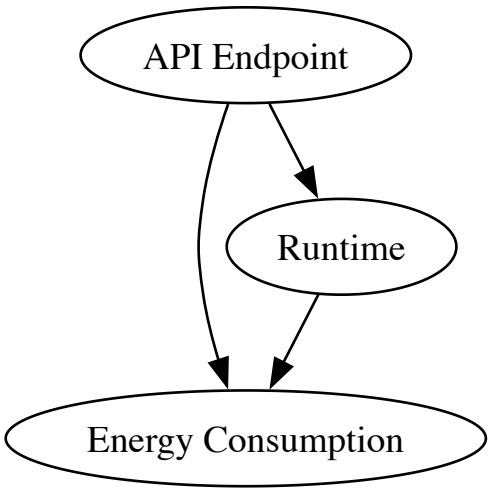
1. Model visualization
2. Data preparation
3. Prior probability selection
4. prior predictive checks
5. Model fitting and sampling
6. Trace plot and summary analysis
7. Shrinkage (Pooling) plots
8. Posterior predictive checks
9. Hypothesis conclusion
10. Counterfactual analysis

1. Model Visualization

First, we draw the model design as a simple directed acyclic graph (DAG).

In [20]:
```
# Initializing a Causal Inference Model (DAG: Directed Acyclic Graph)
CausalGraphicalModel(
    nodes=["Runtime", "API Endpoint", "Energy Consumption"],
    edges=[
        ("Runtime", "Energy Consumption"),
        ("API Endpoint", "Runtime"),
        ("API Endpoint", "Energy Consumption")
    ]
).draw()
```

Out[20]:



The DAG reflects that **API Endpoint** influences **Energy Consumption** both directly and indirectly through **Runtime**. Since **Runtime** and **Energy Consumption** are highly correlated (≈ 0.99)—as mentioned in the casuality analysis—the indirect effect via **Runtime** is likely dominant, while the direct effect of **API Endpoint** on **Energy Consumption** may be minimal but still worth modeling.

2. Data Preparation

First, we are investigating the `endpoint` variable to asses the stratification group of this hypothesis. No baseline category is chosen here due to the nature of H3, where we are comparing across all different endpoints equally. Thus we are just creating some variables for later use and encoding endpoint as a categorical variable.

In [21]:
```
print(f'Number of unique endpoints:
{len(df['endpoint'].unique())}')
print(f'List of unique endpoints: \n{df['endpoint'].unique()}')
```

```
Number of unique endpoints: 14
List of unique endpoints:
['/api/register' '/api/msgs/user0' '/api/msgs' '/api/fllws/user'
 '/api/unfllws/user' '/api/latest' '/register' '/login' '/user/follow'
 '/add_message' '/public' '/user/user0' '/user/unfollow' '/logout']
```

In [22]:
```
# Encode endpoint as a category
df['ep_idx'] = df['endpoint'].astype('category').cat.codes

# Create variable for endpoint index values
endpoint_idx = df['ep_idx'].values

# Create variable for number of endpoints
n_endpoints = df['ep_idx'].nunique()

# Create variable for runtime values
```

```
runtime = df['runtime'].values

# Create variable for the namings of the endpoints
endpoint_categories =
df['endpoint'].astype('category').cat.categories.tolist()
```

3. Prior Probability Selection

The priors are chosen mainly based on domain knowledge and plausible assumptions. They aim to reflect realistic energy-use behavior across API endpoints without encoding specific expectations from the given dataset.

- **Population-Level Mean for Intercepts**: μ_α **~ Normal(0.5, 0.2)**
  Again, mu represents the average baseline energy consumption across API endpoints. We assume a typical API to be at around 0.5 Joules, with moderate uncertainty thus standard deviation of 0.2 (identical to α of H1 and H2).

- **Population-Level Mean for Slopes**: μ_β **~ Normal(3.0, 1.0)**
  Reflects the average increase in energy per second of runtime. The prior is chosen to centers on 3 J/s, suggesting runtime has a positive effect but allows for variability due to the rather significant uncertainty of this estimate.

- **Variability in Intercepts**: σ_α **~ HalfNormal(0.2)**
  Controls how much the baseline energy (the intercepts) varies between endpoints. Allows for slight heterogeneity. So we expect that not all endpoints behave identically but we also do not expect extreme differences across them.

- **Variability in Slopes**: σ_β**~ HalfNormal(1.0)**
  Captures variation in the runtime slope across endpoints which is crucial to this hypothesis. A wider scale allows endpoints to differ in how runtime influences energy.

- **Residual Standard Deviation**: σ**~ HalfNormal(0.25)**
  The same as in H1 and H2. Represents residual variability in energy not explained by the model. The value 0.25 will assume small noise in the measurements.

As such, we can specify our **multilevel model** as:

$$
\begin{aligned}
h_i &\sim \mathcal{N}(\mu_i, \sigma) && [\text{likelihood for observed energy consumption}] \\
\mu_i &= \alpha_{\text{ep}[i]} + \beta_{\text{ep}[i]} \cdot R_i && [\text{linear model with runtime predictor}] \\
\alpha_j &\sim \mathcal{N}(\mu_\alpha, \sigma_\alpha) && [\text{endpoint-specific intercepts}] \\
\beta_j &\sim \mathcal{N}(\mu_\beta, \sigma_\beta) && [\text{endpoint-specific slopes}] \\
\mu_\alpha &\sim \mathcal{N}(0.4, 0.2) && [\text{population-level mean for intercepts}] \\
\mu_\beta &\sim \mathcal{N}(3.0, 1.0) && [\text{population-level mean for slopes}] \\
\sigma_\alpha &\sim \text{HalfNormal}(0.2) && [\text{variability in intercepts}] \\
\sigma_\beta &\sim \text{HalfNormal}(1.0) && [\text{variability in slopes}] \\
\sigma &\sim \text{HalfNormal}(0.25) && [\text{residual standard deviation}]
\end{aligned}
$$

These are mostly quite vague priors, chosen to provide regularization without being overly restrictive.

- We believe they reflect plausible assumptions about energy consumption in API-based applications
- The hierarchical structure also promotes partial pooling in itself, helping prevent overfitting in endpoints with limited data.
- With *prior predictive checks* we will further validate these choices by examining their implications before fitting the model.

4. Prior Predictive Checks

We perform prior predictive checks by sampling data from the priors and plotting the resulting distributions of:

1. Simulated energy consumption values (simulated y)
2. Simulated energy means (simulated μ - for both α and β at population level. So for μ_α and μ_β, respectively)
3. Distributions of each prior parameter (α, β, and σ)

Additionally, we check the amount of negative simulated energy consumption values and energy means, as we would expect these to be positive.

Performing these prior predictive checks allows us to visualize the implications of our prior choices and ensure they are reasonable before fitting the model to the actual data.

In [23]:

```
# Define the model for prior predictive checks
with pm.Model() as model_h3_prior_pred:
    # Hyperpriors for the population-level intercept and slope
    alpha_pop = pm.Normal('alpha_pop', mu=0.5, sigma=.2)  #
Population-level intercept
    beta_pop = pm.Normal('beta_pop', mu=3, sigma=1)       #
Population-level slope

    # Hyperpriors for the variability in intercepts and slopes
```

```
    sigma_alpha = pm.HalfNormal('sigma_alpha', sigma=.2)   # 
Variability in intercepts
    sigma_beta = pm.HalfNormal('sigma_beta', sigma=1)      # 
Variability in slopes

    # Group-level intercepts and slopes for each endpoint
    alpha = pm.Normal('alpha', mu=alpha_pop, sigma=sigma_alpha, 
shape=n_endpoints)
    beta = pm.Normal('beta', mu=beta_pop, sigma=sigma_beta, 
shape=n_endpoints)

    # Model for energy consumption
    mu = alpha[endpoint_idx] + beta[endpoint_idx] * runtime
    sigma = pm.HalfNormal('sigma', sigma=.25)              # 
Residual standard deviation

    # Likelihood
    y_simulated = pm.Normal('y_simulated', mu=mu, sigma=sigma, 
shape = y.shape[0])

    # Sample from the prior predictive distribution
    prior_pred_samples_h3 = 
pm.sample_prior_predictive(samples=1000, random_seed=42)

# Plot the prior predictive distribution for simulated energy 
consumption (y_simulated)
print('Plotting Prior Predictive Distribution for y_simulated 
(Energy Consumption):')
simulated_y = 
prior_pred_samples_h3.prior['y_simulated'].stack(samples = 
('chain', 'draw')).values  # Stack simulated values
az.plot_dist(simulated_y, kind = 'hist', hist_kwargs = {'alpha': 
.8, 'bins': 100})
plt.xlabel('Simulated Energy Consumption (Joules)')
plt.ylabel('Density')
plt.title('Prior Predictive Check: Distribution of Simulated 
Energy Consumption (y_simulated)')
x_llimit, x_ulimit = np.min(simulated_y), np.max(simulated_y)
plt.xlim(x_llimit, x_ulimit)  # x-axis limits
plt.xticks(np.linspace(x_llimit, x_ulimit, num = 15), rotation = 
45)
plt.tight_layout()
```

```
plt.show()

# Plotting the prior predictive distributions for mu_values_alpha
and mu_values_beta side by side
fig, axes = plt.subplots(1, 2, figsize=(18, 6), sharey=True)

# Plot for mu_values_alpha
mu_values_alpha =
prior_pred_samples_h3.prior['alpha_pop'].stack(samples=('chain',
'draw'))
az.plot_dist(mu_values_alpha, kind='hist', hist_kwargs={'alpha':
0.8, 'bins': 50}, ax=axes[0])
axes[0].set_xlabel('Simulated Mean Energy Consumption (Joules)')
axes[0].set_ylabel('Density')
axes[0].set_title('Prior Predictive Check: Alpha
(mu_values_alpha)')
x_min_alpha, x_max_alpha = np.min(mu_values_alpha),
np.max(mu_values_alpha)
axes[0].set_xticks(np.linspace(x_min_alpha, x_max_alpha, num=15))
axes[0].tick_params(axis='x', rotation=45)

# Plot for mu_values_beta
mu_values_beta =
prior_pred_samples_h3.prior['beta_pop'].stack(samples=('chain',
'draw'))
az.plot_dist(mu_values_beta, kind='hist', hist_kwargs={'alpha':
0.8, 'bins': 50}, ax=axes[1])
axes[1].set_xlabel('Simulated Mean Energy Consumption (Joules)')
axes[1].set_title('Prior Predictive Check: Beta
(mu_values_beta)')
x_min_beta, x_max_beta = np.min(mu_values_beta),
np.max(mu_values_beta)
axes[1].set_xticks(np.linspace(x_min_beta, x_max_beta, num=15))
axes[1].tick_params(axis='x', rotation=45)

# Adjust layout
plt.tight_layout()
plt.show()


# Plotting the distributions of the priors themselves for alpha,
betas, and sigma
```

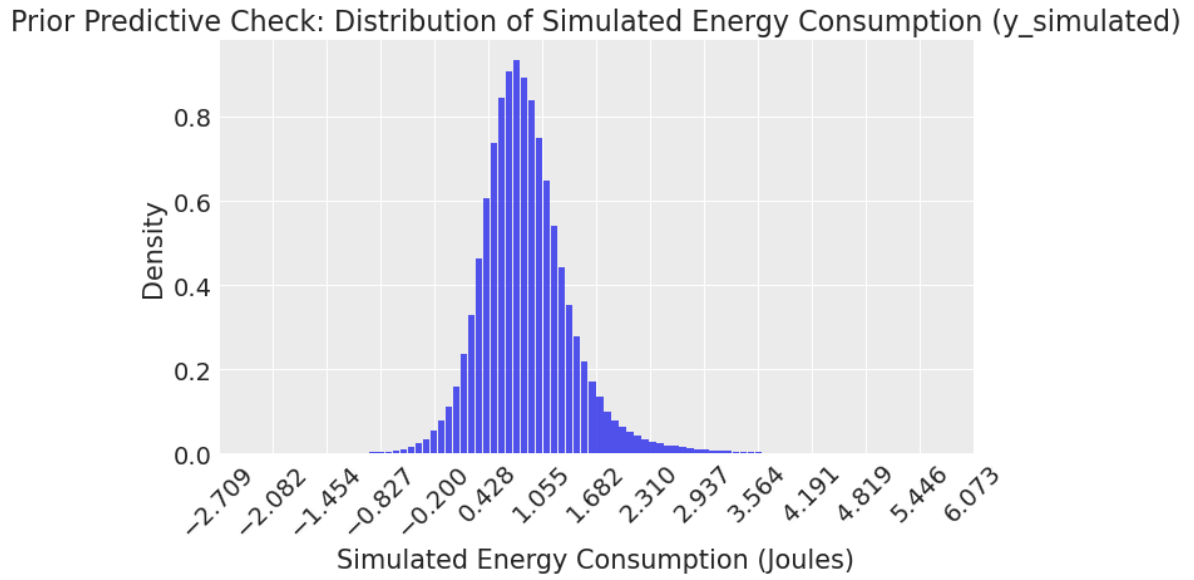
```
print('\nPlotting Prior Distributions for Parameters:')
fig, axes = plt.subplots(1, 3, figsize = (18, 5))

# Alphas
alpha_values = prior_pred_samples_h3.prior['alpha'].stack(samples
= ('chain', 'draw'))
az.plot_dist(alpha_values, ax = axes[0], label = 'alpha', kind =
'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks = np.linspace(np.min(alpha_values), np.max(alpha_values),
num = 15)
axes[0].set_xticks(x_ticks)
axes[0].set_xticklabels(np.round(x_ticks, 2), rotation = 45)
axes[0].set_title('Prior Distribution for alpha')
axes[0].set_xlabel('alpha value')
axes[0].set_ylabel('Density')

# Betas
beta_values = prior_pred_samples_h3.prior['beta'].stack(samples =
('chain', 'draw'))
az.plot_dist(beta_values, ax = axes[1], label = 'betas', kind =
'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks = np.linspace(np.min(beta_values), np.max(beta_values),
num = 15)
axes[1].set_xticks(x_ticks)
axes[1].set_xticklabels(np.round(x_ticks, 2), rotation = 45)
axes[1].set_title('Prior Distribution for beta')
axes[1].set_xlabel('beta value')

# Sigmas
sigma_values = prior_pred_samples_h3.prior['sigma'].stack(samples
= ('chain', 'draw'))
az.plot_dist(sigma_values, ax = axes[2], label = 'sigma', kind =
'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks = np.linspace(np.min(sigma_values), np.max(sigma_values),
num = 15)
axes[2].set_xticks(x_ticks)
axes[2].set_xticklabels(np.round(x_ticks, 2), rotation = 45)
axes[2].set_title('Prior Distribution for sigma')
axes[2].set_xlabel('sigma value')

plt.tight_layout()
plt.show()
```

```
# Check for negative energy values for h3
simulated_y_flat =
prior_pred_samples_h3.prior['y_simulated'].stack(samples=
('chain', 'draw')).values
negative_y_percentage = np.mean(simulated_y_flat < 0) * 100
print(f'\nPercentage of simulated energy consumption values
(y_simulated) < 0: {negative_y_percentage:.2f}%')

simulated_mu_flat =
prior_pred_samples_h3.prior['alpha_pop'].stack(samples=('chain',
'draw')).values
negative_mu_alpha_percentage = np.mean(simulated_mu_flat < 0) *
100
print(f'Percentage of simulated mean energy values (alpha_pop) <
0: {negative_mu_alpha_percentage:.2f}%')

simulated_mu_flat =
prior_pred_samples_h3.prior['beta_pop'].stack(samples=('chain',
'draw')).values
negative_mu_beta_percentage = np.mean(simulated_mu_flat < 0) *
100
print(f'Percentage of simulated mean energy values (beta_pop) <
0: {negative_mu_beta_percentage:.2f}%')
```

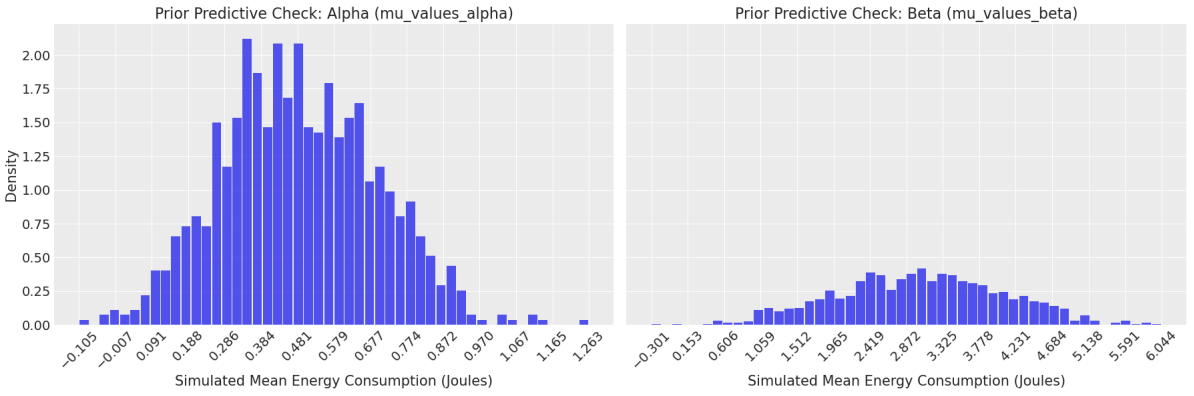
```
Sampling: [alpha, alpha_pop, beta, beta_pop, sigma, sigma_alpha, sigma_bet
a, y_simulated]
```

```
Plotting Prior Predictive Distribution for y_simulated (Energy Consumptio
n):
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/315402559
5.py:35: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```

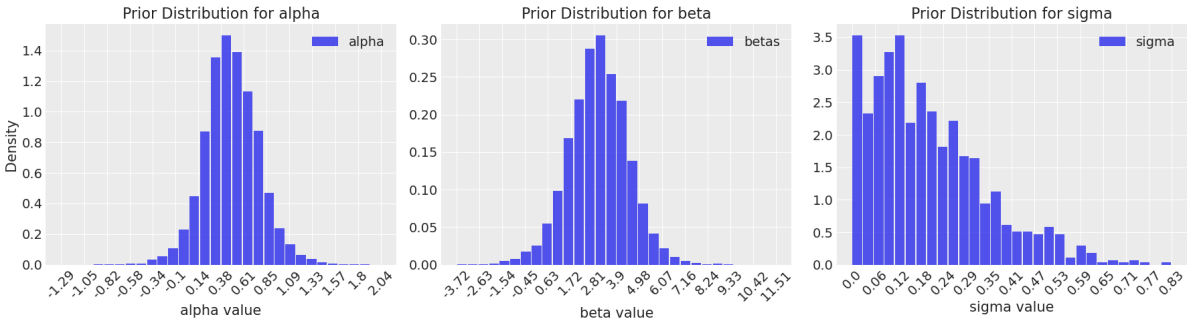



```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/3154025595.py:61: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```




```
Plotting Prior Distributions for Parameters:
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/3154025595.py:97: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```




```
Percentage of simulated energy consumption values (y_simulated) < 0: 2.92%
Percentage of simulated mean energy values (alpha_pop) < 0: 0.60%
Percentage of simulated mean energy values (beta_pop) < 0: 0.10%
```

Interpretation of Prior Predictive Checks for H3

The plots generated by the prior predictive checks in the cell above help us to better understand the implications of the chosen priors:

1. **Distribution of Simulated Energy Consumption (`y_simulated`)**: The simulated energy consumption values align well with plausible energy levels, and look similar to the ones from H1 and H2. The range of simulated values is thus consistent with

what we might intuitively expect for energy consumption in this context. The proportion of negative values is satisfactory small at 2.92%, indicating that the priors constrain the simulated values to even more realistic ranges than in H1 and H2

2. **Distribution of Simulated Mean Energy Consumption (`mu_values`)**: The simulated mean energy consumption values for both μ_α and μ_β are mostly positive, with only .6% and .1% negative values, respectively. This demonstrates that the priors for mean energy consumption are even more well-calibrated and that the simulated values are centered around plausible energy levels, that are realistic in a real world setting.

3. **Prior Distributions for Parameters (α, β, σ)**: The prior distribution plots seem to follow the specified prior distribution parameters well.

5. Model Fitting and Sampling

We fit the multilevel Bayesian model using `PyMC` and sample from the posterior distribution with the No-U-Turn Sampler (NUTS), a Hamiltonian Monte Carlo (HMC) algorithm. The consists of varying intercepts and slopes for each API endpoint, allowing us to model heterogeneous relationships between runtime and energy consumption.

We specify hierarchical priors:

`alpha_pop` and `beta_pop` define the population-level mean intercept and slope, respectively.

`sigma_alpha` and `sigma_beta` define the variability across endpoints, capturing endpoint-level deviations from the population mean.

Each API endpoint is assigned its own α (intercept) and β (slope), drawn from the population distributions. This enables partial pooling, allowing information sharing across endpoints.

We again set the number of samples to 2000 and the number of tuning steps to 1000 with `target_accept=0.95` to ensure a higher acceptance rate for the NUTS sampler, improving convergence and reducing divergences.

We also use `return_inferencedata=True` for compatibility with ArviZ-based diagnostics and enable log_likelihood tracking for model comparison.

A graphical representation of the model structure is visualized using `pm.model_to_graphviz(model_h3)`, which shows the dependency relationships between parameters and observations.

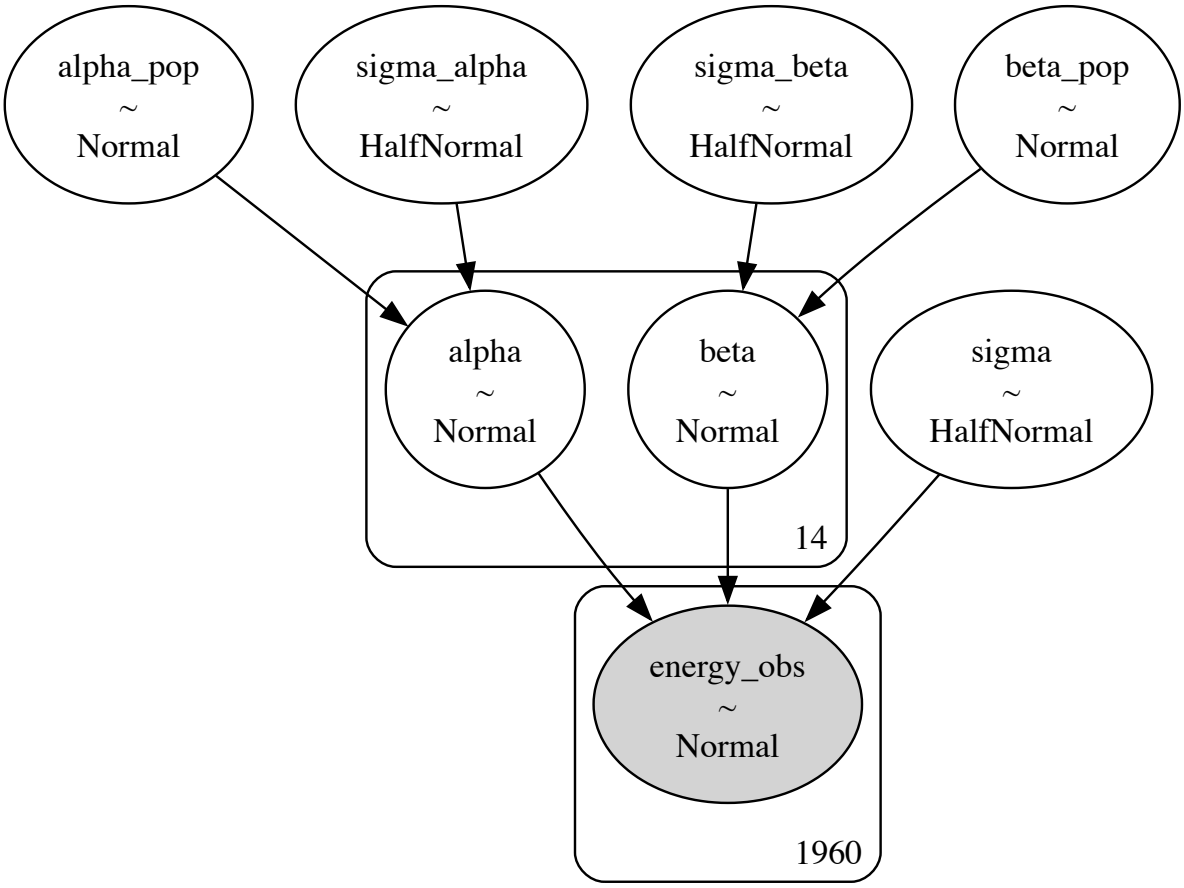
In [ ]:
```
# Defining Bayesian model for H3
with pm.Model() as model_h3:
```

```
    # Hyperpriors for the population-level intercept and slope
    alpha_pop = pm.Normal("alpha_pop", mu=0.5, sigma=.2)   #
Population-level intercept
    beta_pop = pm.Normal("beta_pop", mu=3, sigma=1)        #
Population-level slope

    # Hyperpriors for the variability in intercepts and slopes
    sigma_alpha = pm.HalfNormal("sigma_alpha", sigma=.2)   #
Variability in intercepts
    sigma_beta = pm.HalfNormal("sigma_beta", sigma=1)      #
Variability in slopes

    # Group-level intercepts and slopes for each endpoint
    alpha = pm.Normal("alpha", mu=alpha_pop, sigma=sigma_alpha,
shape=n_endpoints)
    beta = pm.Normal("beta", mu=beta_pop, sigma=sigma_beta,
shape=n_endpoints)

    # Model for energy consumption
    mu = alpha[endpoint_idx] + beta[endpoint_idx] * runtime
    sigma = pm.HalfNormal("sigma", sigma=.25)              #
Residual standard deviation

    # Likelihood
    energy_obs = pm.Normal("energy_obs", mu=mu, sigma=sigma,
observed=y)

    # Sampling
    trace_h3 = pm.sample(2000, tune=1000, target_accept=0.95,
return_inferencedata=True, random_seed=42, idata_kwargs=
{"log_likelihood": True})

pm.model_to_graphviz(model_h3)
```

```
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [alpha_pop, beta_pop, sigma_alpha, sigma_beta, alpha, beta, sigma]
Output()
```

```
Sampling 4 chains for 1_000 tune and 2_000 draw iterations (4_000 + 8_000 d
raws total) took 36 seconds.
```

Out[ ]:



6. Trace Plot and Summary Analysis

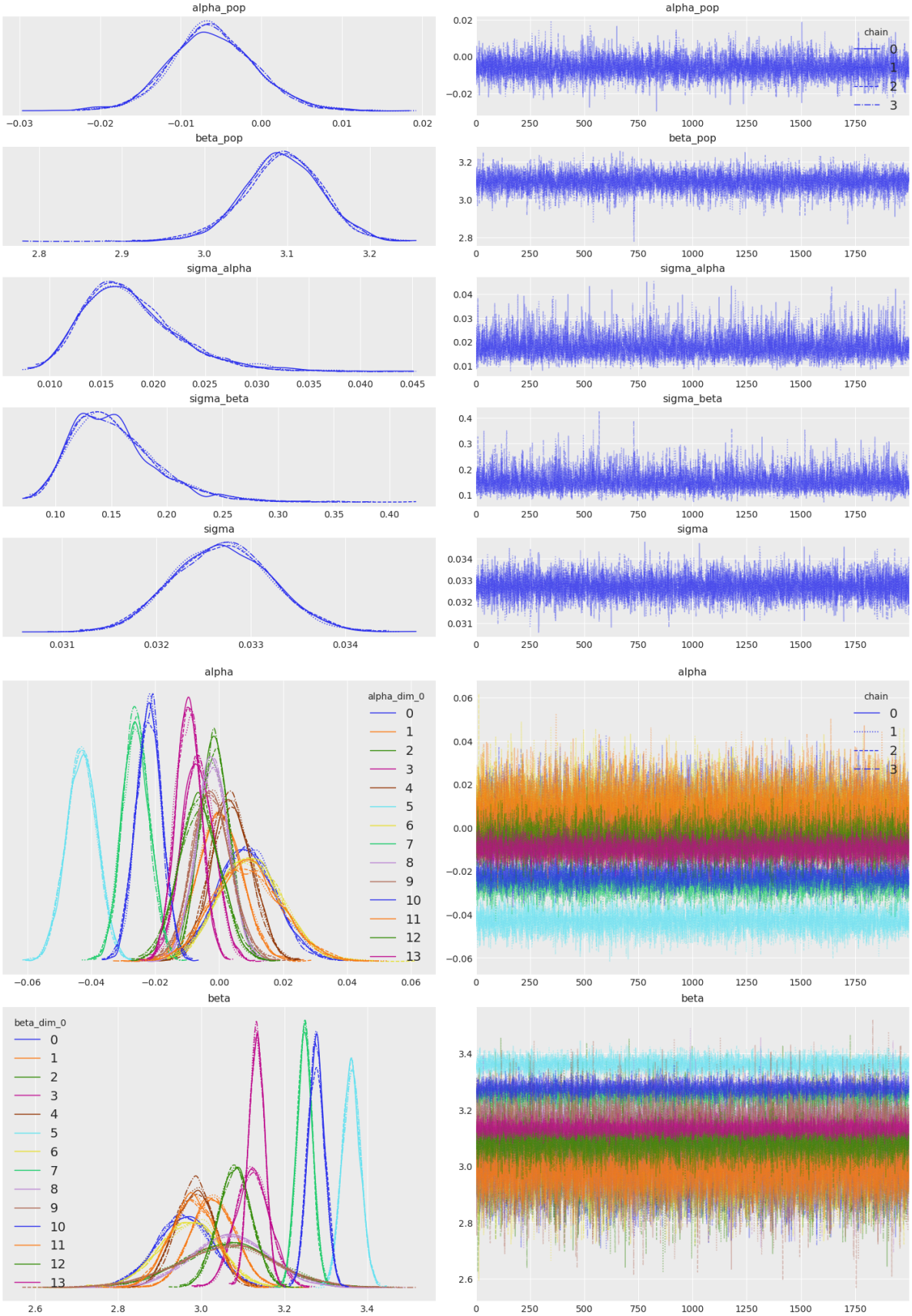
We use **trace plots** to visualize the posterior distributions generated by the four chains in the sampling process. We use **summary statistics** to assess the convergence, quality of estimates, and statistical significance.

In [ ]:

```
# Posterior analysis
az.plot_trace(trace_h3, var_names = ['alpha_pop', 'beta_pop',
'sigma_alpha', 'sigma_beta', 'sigma'], figsize = (14, 10), legend
= True)
az.plot_trace(trace_h3, var_names = ['alpha', 'beta'], figsize =
(14, 10), legend = True)
az.summary(trace_h3, var_names=['alpha_pop', 'beta_pop',
'sigma_alpha', 'sigma_beta', 'sigma', 'alpha', 'beta'],
round_to=4)
```

Out[ ]:

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_
alpha_pop	-0.0060	0.0054	-0.0165	0.0037	0.0001	0.0000	7533.5764	5663.8
beta_pop	3.0939	0.0471	3.0083	3.1863	0.0006	0.0004	7138.4078	5756.8
sigma_alpha	0.0179	0.0047	0.0102	0.0270	0.0001	0.0000	6140.0319	5971.3
sigma_beta	0.1529	0.0389	0.0895	0.2263	0.0005	0.0004	5807.3250	5546.9
sigma	0.0327	0.0005	0.0317	0.0336	0.0000	0.0000	12521.4435	5872.0
alpha[0]	0.0086	0.0090	-0.0081	0.0259	0.0001	0.0001	6273.7409	5727.0
alpha[1]	-0.0000	0.0070	-0.0131	0.0129	0.0001	0.0001	7730.7694	5548.1
alpha[2]	-0.0014	0.0048	-0.0108	0.0071	0.0001	0.0000	8192.0550	6328.1
alpha[3]	-0.0067	0.0050	-0.0162	0.0024	0.0001	0.0000	8108.7282	6146.0
alpha[4]	0.0038	0.0063	-0.0080	0.0157	0.0001	0.0001	7627.4056	6332.8
alpha[5]	-0.0430	0.0048	-0.0525	-0.0344	0.0001	0.0000	7472.6186	5907.1
alpha[6]	0.0099	0.0100	-0.0084	0.0287	0.0001	0.0001	6288.9567	5628.8
alpha[7]	-0.0261	0.0042	-0.0337	-0.0178	0.0000	0.0000	8239.0201	5955.6
alpha[8]	-0.0020	0.0052	-0.0116	0.0079	0.0001	0.0001	7332.5393	5995.4
alpha[9]	-0.0025	0.0061	-0.0140	0.0090	0.0001	0.0001	7691.5171	5599.8
alpha[10]	-0.0220	0.0041	-0.0297	-0.0144	0.0000	0.0000	8900.5677	6197.3
alpha[11]	0.0087	0.0106	-0.0111	0.0284	0.0001	0.0001	5267.6938	5442.5
alpha[12]	-0.0062	0.0065	-0.0183	0.0061	0.0001	0.0001	8223.8094	6110.7
alpha[13]	-0.0097	0.0040	-0.0168	-0.0019	0.0000	0.0000	8375.8459	6344.1
beta[0]	2.9576	0.0679	2.8334	3.0901	0.0008	0.0006	6398.3792	5707.2
beta[1]	3.0278	0.0538	2.9289	3.1298	0.0006	0.0004	7675.8260	5578.0
beta[2]	3.0625	0.1133	2.8485	3.2707	0.0013	0.0009	7964.5442	6095.0
beta[3]	3.1232	0.0410	3.0471	3.2006	0.0005	0.0003	7917.5939	6624.2
beta[4]	2.9845	0.0480	2.8938	3.0734	0.0005	0.0004	7723.5042	6092.4
beta[5]	3.3610	0.0221	3.3197	3.4021	0.0002	0.0002	7958.0658	6134.7
beta[6]	2.9693	0.0735	2.8303	3.1060	0.0009	0.0007	6214.3936	5570.
beta[7]	3.2490	0.0192	3.2140	3.2865	0.0002	0.0001	8550.0293	5990.
beta[8]	3.0682	0.0944	2.8852	3.2367	0.0011	0.0008	7373.2140	5753.5
beta[9]	3.0588	0.1191	2.8286	3.2809	0.0014	0.0010	7434.8576	5956.2
beta[10]	3.2766	0.0204	3.2383	3.3146	0.0002	0.0002	8527.2271	5672.5
beta[11]	2.9784	0.0536	2.8718	3.0731	0.0007	0.0005	5335.8503	5568.2
beta[12]	3.0806	0.0405	3.0028	3.1551	0.0005	0.0003	7828.8443	5830.9
beta[13]	3.1335	0.0191	3.0992	3.1701	0.0002	0.0001	8422.1729	6447.1



Interpretation of Trace Plots and Summary Statistics for H3

1. **Summary Table**

- The `r_hat` values for all parameters are very close to 1.0, which indicates good convergence.
- All `ess_bulk` and `ess_tail` values are consistently above 1000, this indicates efficient sampling by the NUTS sampler and high reliability of posterior summaries.

- The `HDI` for the slopes (β) does not include 0, providing strong evidence of a positive relationship between runtime and energy consumption. However, for some intercepts (α), the `HDI` includes 0. This suggests that for some specific endpoints, there isn't enough evidence in the data to conclude that the energy consumption at zero rumtime is definitively non-zero.

1. **Posterior Distributions**

- The posterior distributions for the population-level parameters (`alpha_pop` , `beta_pop` , `sigma_alpha` , `sigma_beta` , `sigma`) are unimodal and do not appear overly wide or flat, suggesting that the model has learned reasonable estimates for these parameters.
- The posterior distributions for the endpoint-specific parameters (`alpha` , `beta`) show the range of plausible values for each endpoint's intercept and slope.

1. **Trace Plots**

- Trace plots for `alpha_pop` , `beta_pop` , `sigma_alpha` , `sigma_beta` , and `sigma` are rather stationary and without trends, indicating that the sampling chains have converged and are exploring the posterior distribution well. Only small deviations in the sampling trace.
- The trace plots for `alpha` (endpoint-specific intercepts) show variation across endpoints, suggesting that the baseline energy consumption differs depending on the API endpoint.
- The trace plots for `beta` (endpoint-specific slopes) also show variation, indicating that the impact of runtime on energy consumption varies across different API endpoints. Some endpoints may have a steeper slope where runtime has a stronger impact, while others have a more shallow slope with runtime hhaving a weaker impact.

7. Shrinkage (Pooling) Plots

Shrinkage plots, sometimes called *pooling plots*, are a useful visualization tool in hierarchical, multilevel models to illustrate the relationship between group-level parameters (here: endpoint-specific intercepts and slopes) and population-level parameters. In the context of **H3**, these plots help us understand how the endpoint-specific intercepts (α_j) and slopes (β_j) for runtime are influenced by the population-level intercept (α_{pop}) and slope (β_{pop}).

Intercepts (α_j)

The first shrinkage plot shows the endpoint-specific intercepts (α_j) along with their 95% credible intervals. These intercepts represent the baseline energy consumption for each API endpoint. The horizontal red dashed line indicates the population-level intercept (α_{pop}), which represents the average baseline energy consumption across all endpoints.

Shrinkage occurs because endpoints with sparse data are pulled closer to the population-level mean, reflecting the partial pooling effect of the hierarchical model.

Slopes (β_j)

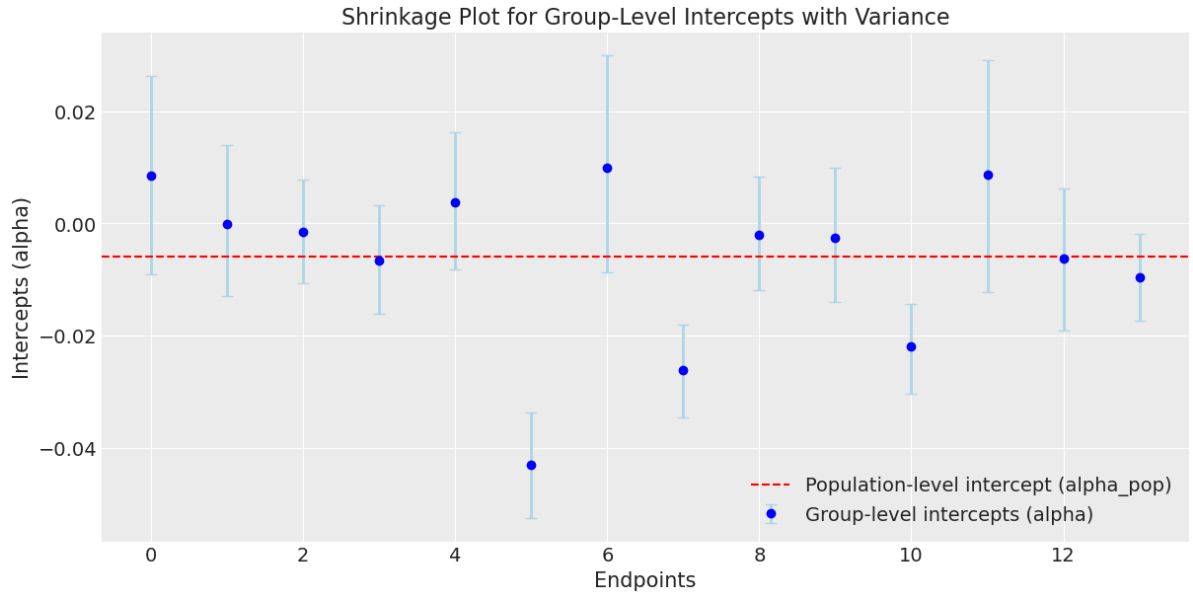
Our second shrinkage plot visualizes the endpoint-specific slopes (β_j) for runtime, along with their 95% credible intervals. These slopes represent the effect of runtime on energy consumption for each API endpoint. The horizontal red dashed line indicates the population-level slope (β_{pop}), which represents the average effect of runtime across all endpoints. Similar to the intercepts, endpoints with less data exhibit stronger shrinkage toward the population-level slope.

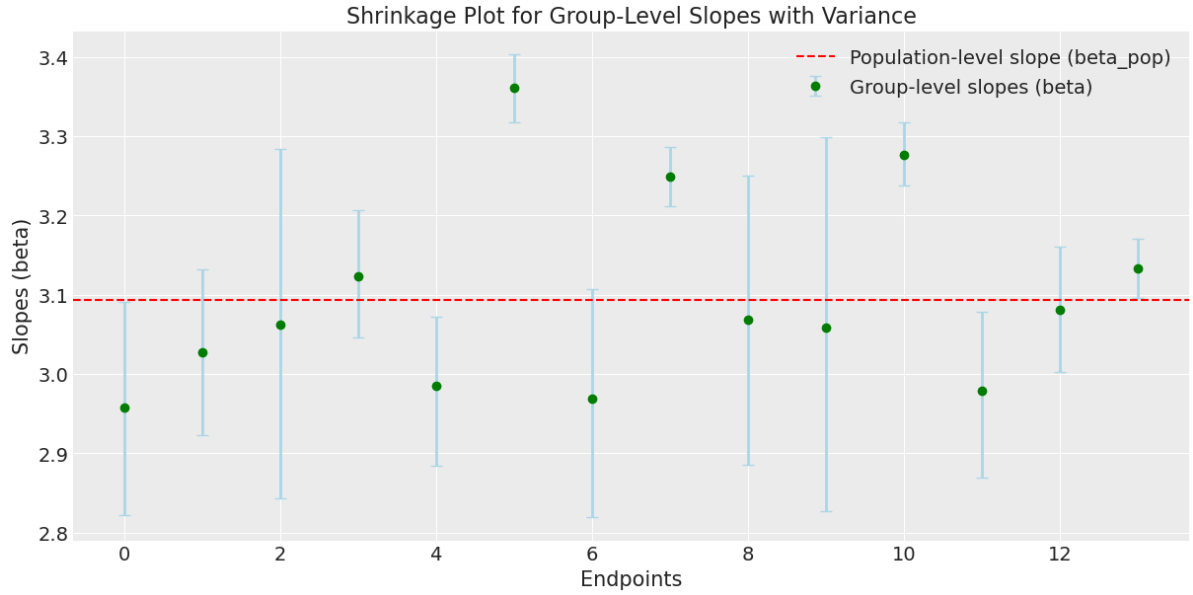
In [27]:

```
import matplotlib.pyplot as plt
import arviz as az
import numpy as np

# Extract posterior means and credible intervals for population-
level and group-level parameters
alpha_pop_mean = trace_h3.posterior["alpha_pop"].mean().item()
alpha_means = trace_h3.posterior["alpha"].mean(dim=["chain",
"draw"]).values
alpha_hdi = az.hdi(trace_h3.posterior["alpha"], hdi_prob=0.95) #
95% credible intervals

lower = alpha_hdi["alpha"].sel(hdi="lower").values
upper = alpha_hdi["alpha"].sel(hdi="higher").values

endpoints = np.arange(len(alpha_means))  # x-axis: endpoint
indices

# Create the shrinkage plot
plt.figure(figsize=(12, 6))
# Plot group-level intercepts with error bars (credible
intervals)
plt.errorbar(
    endpoints,
    alpha_means,
    yerr=[alpha_means - lower, upper - alpha_means],
    fmt="o",
    color="blue",
    ecolor="lightblue",
    elinewidth=2,
    capsize=4,
```

```
    label="Group-level intercepts (alpha)",
)
# Add a horizontal line for the population-level intercept
plt.axhline(alpha_pop_mean, color="red", linestyle="--",
label="Population-level intercept (alpha_pop)")

# Add labels, title, and legend
plt.xlabel("Endpoints")
plt.ylabel("Intercepts (alpha)")
plt.title("Shrinkage Plot for Group-Level Intercepts with
Variance")
plt.legend()
plt.grid(True)
plt.show()

#####################################################################

# Extract posterior means and credible intervals for population-
level and group-level parameters
beta_pop_mean = trace_h3.posterior["beta_pop"].mean().item()
beta_means = trace_h3.posterior["beta"].mean(dim=["chain",
"draw"]).values
beta_hdi = az.hdi(trace_h3.posterior["beta"], hdi_prob=0.95) #
95% credible intervals

lower = beta_hdi["beta"].sel(hdi="lower").values
upper = beta_hdi["beta"].sel(hdi="higher").values

endpoints = np.arange(len(beta_means))  # x-axis: endpoint
indices

# Create the shrinkage plot
plt.figure(figsize=(12, 6))
# Plot group-level intercepts with error bars (credible
intervals)
plt.errorbar(
    endpoints,
    beta_means,
    yerr=[beta_means - lower, upper - beta_means],
    fmt="o",
    color="green",
    ecolor="lightblue",
```

```
    elinewidth=2,
    capsize=4,
    label="Group-level slopes (beta)",
)
# Add a horizontal line for the population-level intercept
plt.axhline(beta_pop_mean, color="red", linestyle="--",
label="Population-level slope (beta_pop)")

# Add labels, title, and legend
plt.xlabel("Endpoints")
plt.ylabel("Slopes (beta)")
plt.title("Shrinkage Plot for Group-Level Slopes with Variance")
plt.legend()
plt.grid(True)
plt.show()
```





Interpretation of Shrinkage Plots for **H3**

These shrinkage plots are particularly relevant for **H3**, as they allow us to assess the variability in runtime's effect on energy consumption across endpoints. If the endpoint-specific slopes (β_j) vary significantly, this supports the hypothesis that runtime has a stronger impact on energy consumption for some API endpoints than others. Conversely, if the slopes are tightly clustered around the population-level slope (β_{pop}), it suggests that runtime's effect is consistent across endpoints.

And as it can be seen from the shrinkage plots, the intercepts and slopes vary quite a lot for the different endpoints. The α's are slightly more closely aligned with the population-level intercept than the β's are for the population level slope. The is also a generally higher variance in the endpoint-specific β's, which all in all suggests, that runtime's effect differs across endpoints.

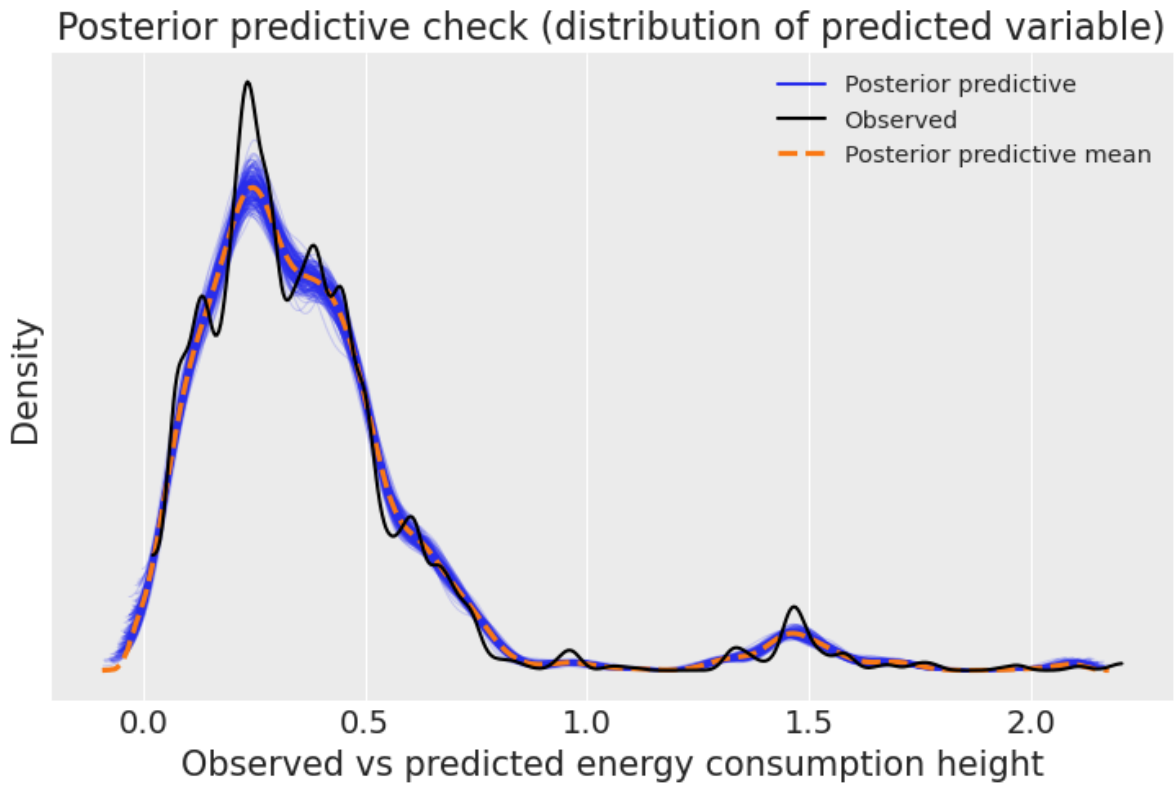
8. Posterior Predictive Checks

We perform posterior predictive checks to assess the model's fit to the data. By sampling from the posterior distribution, we can compare the observed data to the predicted responses. We plot the observed energy consumption values against the predicted values.

In [28]:
```
#Generate posterior predictive samples
with model_h3:
    ppc_h3 = pm.sample_posterior_predictive(trace_h3, var_names =
['energy_obs'], random_seed = 42)

_, ax = plt.subplots()
az.plot_ppc(ppc_h3, num_pp_samples = 200, ax = ax)
ax.set_xlabel('Observed vs predicted energy consumption height')
ax.set_ylabel('Density')
ax.set_title('Posterior predictive check (distribution of
predicted variable)')
```

```
Sampling: [energy_obs]
Output()
```

Out[28]:
```
Text(0.5, 1.0, 'Posterior predictive check (distribution of predicted varia
ble)')
```



Interpretation of Posterior Predictive Checks for H3

The plot once again shows the observed energy consumption values against the predictions from the posterior predictive checks. We see that the predictions fit the observed data much better than in the two previous models (for H1 and H2). It even captures the small "hill" in the observed energy consumption at around 1.5 Joules on the x-axis. Furthermore, the predictions almost produce no negative values anymore, which is plausible.

9. Hypothesis Conclusion

While the trace and shrinkage plots indicate that runtime's effect on energy consumption differs for specific endpoint, we still intent to test H3 more statistically.

To do so, we will implement close to the same method as from H1 and H2, however now we quantify the evidence for each endpoint by computing the proportion of posterior samples where $\beta_i > \beta_j$. It gives the posterior probability $P(\beta_i > \beta_j|\text{data})$. This is done for all 182 combinations $(14 * 14 - 14)$ of comparisions between the 14 endpoints.

The results are shown in the pandas DataFrame below.

In [30]:
```
# Extract posterior samples for betas (endpoint-specific slopes)
posterior_betas = trace_h3.posterior['beta']  # Shape: (chains,
draws, num_endpoints)

# Calculate pairwise probabilities for each endpoint
num_endpoints = posterior_betas.shape[-1]
```

```
prob_beta_greater = np.zeros((num_endpoints, num_endpoints))

for i in range(num_endpoints):
    for j in range(num_endpoints):
        if i != j:
            # Probability that beta for endpoint i is greater
than beta for endpoint j
            prob_beta_greater[i, j] = (posterior_betas[..., i] >
posterior_betas[..., j]).mean()

# Create a list to store the pairwise probabilities
pairwise_probs = []

# Loop through all endpoint pairs and store the results
for i, name_i in enumerate(endpoint_categories):
    for j, name_j in enumerate(endpoint_categories):
        if i != j:
            pairwise_probs.append({
                "Endpoint_i": name_i,
                "Endpoint_j": name_j,
                "P(beta_i > beta_j)": prob_beta_greater[i, j]
            })

# Convert the list to a pandas DataFrame
pairwise_probs_df = pd.DataFrame(pairwise_probs)

# Display the DataFrame
pairwise_probs_df
```

Out[30]:

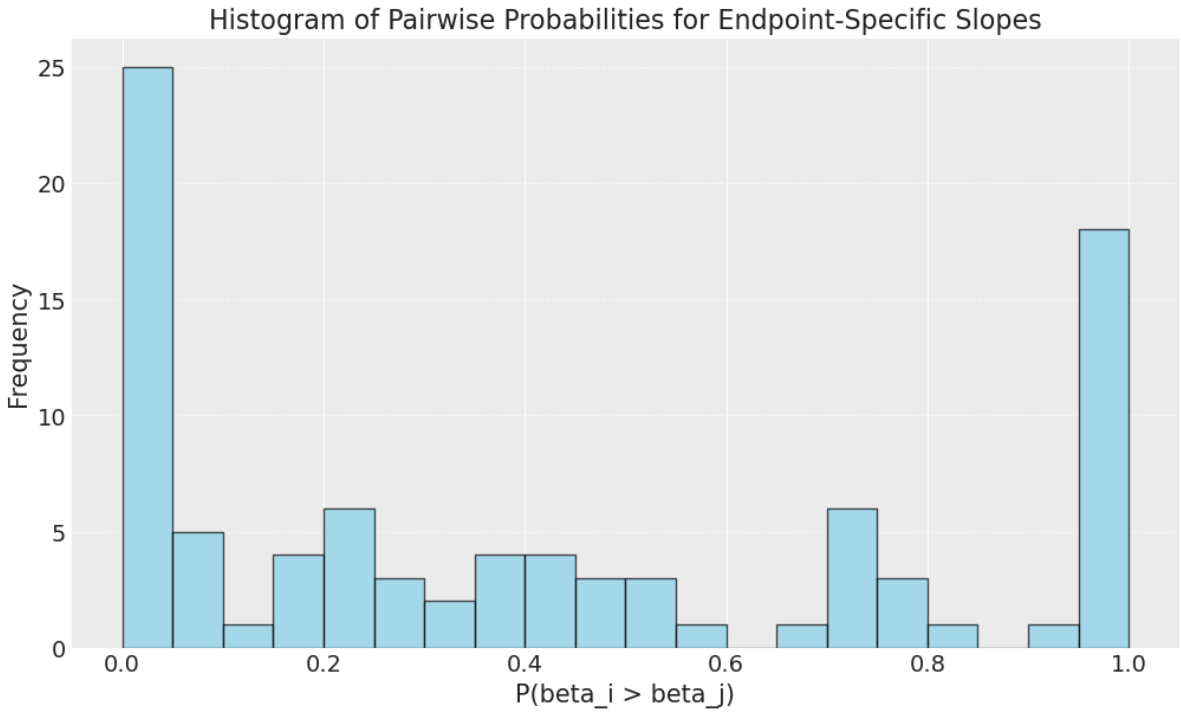
	Endpoint_i	Endpoint_j	P(beta_i > beta_j)
0	/add_message	/api/fllws/user	0.204125
1	/add_message	/api/latest	0.197000
2	/add_message	/api/msgs	0.016375
3	/add_message	/api/msgs/user0	0.376375
4	/add_message	/api/register	0.000000
...	...	...	...
177	/user/user0	/logout	0.750875
178	/user/user0	/public	0.737000
179	/user/user0	/register	0.000000
180	/user/user0	/user/follow	0.997500
181	/user/user0	/user/unfollow	0.884250

182 rows × 3 columns

By quickly glancing over the df, it is clear that the $P(\beta_i > \beta_j|\text{data})$'s differ drastically with both high and very low posterior probabilities. To get a better overview we blot a histogram of the df below:

In [31]:

```
# Flatten the pairwise probabilities into a 1D array
prob_beta_greater_flat =
prob_beta_greater[np.triu_indices(num_endpoints, k=1)]  # Use
only upper triangle to avoid duplicates

# Plot a histogram of the pairwise probabilities
plt.figure(figsize=(10, 6))
plt.hist(prob_beta_greater_flat, bins=20, color='skyblue',
edgecolor='black', alpha=0.7)
plt.xlabel("P(beta_i > beta_j)")
plt.ylabel("Frequency")
plt.title("Histogram of Pairwise Probabilities for Endpoint-
Specific Slopes")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



This plot further supports the claim, that the posterior probabilities differ a lot when comparing the different endpoints. Most posterior probabilites lay close to 0 or 1, indicating that they actually differ *a lot* from each other. For the runtime to have a similar impact on energy consumption for all endpoints, the histogram should peak at around 0.5.

As such, we can conclude that the model provides strong evidence supporting H3, i.e., runtime has a stronger impact on energy consumption for some API endpoints than others. The posterior probabilities indicate large differences between endpoints.

The trace and shrinkage plots along with the summary statistics further support this: all chains mix well, `R–hat` values are close to 1, and effective sample sizes are high, indicating good convergence and reliable estimates. Posterior predictive checks demonstrate that the model captures the observed data distribution almost exceptionally well with minor deviations. Taken together, these results provide compelling evidence in support of H3.

10. Counterfactual analysis

We finally seek to perform a counterfactual analysis within our model for H3, as an extra layer of analysis. We define an "extreme" value for runtime, multiplying the .max() value in the dataset with 10. For each endpoint we then compute the counterfactual energy predictions and plot these posterior predictions for the much larger chosen runtime value. We further create summary statistics for the latter.

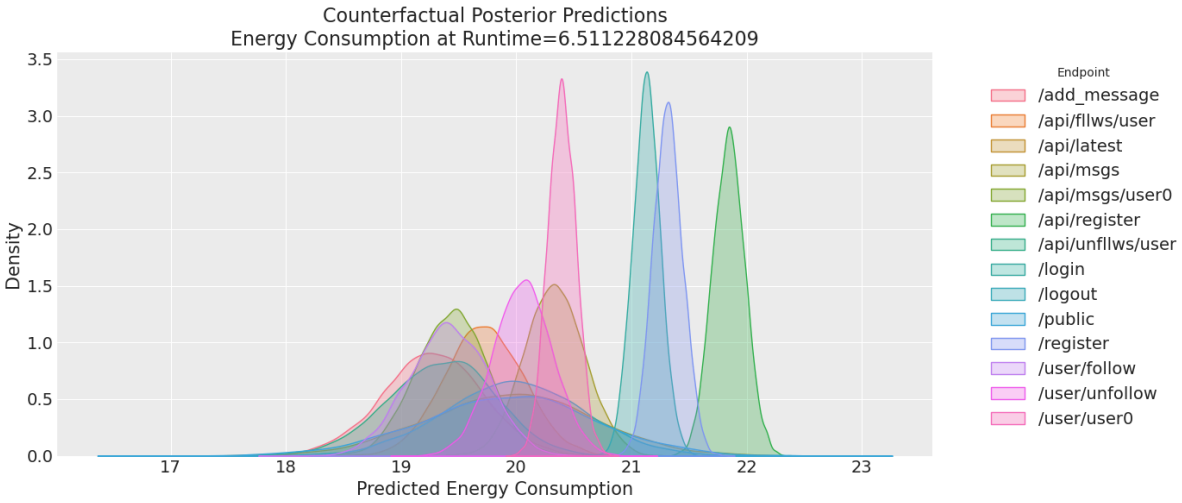
In [32]:
```
# Max observed runtime
print(runtime.max())
```

```
0.6511228084564209
```

In [33]:
```
runtime_cf = np.array([runtime.max() * 10])  #increasing the
runtime by 10

# Number of posterior samples
n_samples = trace_h3.posterior.sizes['draw'] *
trace_h3.posterior.sizes['chain']

# Get posterior samples of alpha and beta for all endpoints
alpha_samples = trace_h3.posterior['alpha'].stack(samples=
('chain', 'draw')).values  # shape: (n_endpoints, n_samples)
beta_samples = trace_h3.posterior['beta'].stack(samples=('chain',
'draw')).values    # same shape

# Expand counterfactual runtime to match samples
runtime_cf_expanded = runtime_cf[0]

# Compute counterfactual energy predictions for each endpoint
mu_cf = alpha_samples + beta_samples * runtime_cf_expanded
# shape: (n_endpoints, n_samples)
```

In [34]:
```
# Plot
plt.figure(figsize=(14, 6))
endpoint_labels = endpoint_categories
colors = sns.color_palette('husl', n_endpoints)

for i in range(n_endpoints):
    sns.kdeplot(mu_cf[i, :], label=f'{endpoint_labels[i]}',
color=colors[i], fill=True, alpha=0.3)

plt.title(f'Counterfactual Posterior Predictions\nEnergy
Consumption at Runtime={runtime_cf[0]}')
plt.xlabel('Predicted Energy Consumption')
plt.ylabel('Density')
plt.legend(title='Endpoint', bbox_to_anchor=(1.05, 1), loc='upper
left')
plt.tight_layout()
plt.show()
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/347761027
1.py:13: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



In [35]:
```
summary_stats = pd.DataFrame({
    'endpoint': endpoint_labels,
    'mean_energy': mu_cf.mean(axis=1),
    'std_energy': mu_cf.std(axis=1),
    '5%': np.percentile(mu_cf, 5, axis=1),
    '95%': np.percentile(mu_cf, 95, axis=1),
})

print(summary_stats.sort_values('mean_energy', ascending=False))
```

	endpoint	mean_energy	std_energy	5%	95%
5	/api/register	21.840920	0.139624	21.611732	22.070199
10	/register	21.312905	0.129650	21.102048	21.530230
7	/login	21.128948	0.121658	20.928398	21.330866
13	/user/user0	20.393254	0.121261	20.196328	20.592612
3	/api/msgs	20.329341	0.262472	19.906373	20.766059
12	/user/unfollow	20.052485	0.258134	19.624218	20.474508
8	/logout	19.975821	0.610240	18.958727	20.959712
2	/api/latest	19.939326	0.733819	18.732758	21.126973
9	/public	19.913829	0.770007	18.632038	21.170847
1	/api/fllws/user	19.714975	0.343756	19.156196	20.277215
4	/api/msgs/user0	19.436869	0.306808	18.924362	19.934326
11	/user/follow	19.401428	0.339007	18.836920	19.944427
6	/api/unfllws/user	19.343712	0.469185	18.553640	20.088106
0	/add_message	19.266447	0.433279	18.537038	19.976329

Conclusions for Counterfactual Analysis

The analysis demonstrates that the predicted energy consumption varies substantially across endpoints, even though the runtime (extended) is the same for all. This is a direct consequence of the varying slopes (β) and intercepts (α) learned by the hierarchical, multilevel model for each endpoint.

The range of the predicted mean energy spans from ≈ 19.3 to 21.8, so over 2.5 Joules difference. That is a substantial difference, especially when input is held constant.

Some endpoints like "/api/register", "/register" or "/login" have higher mean energy which means their slope (the β)is most likely steeper, which indicates runtime impacting their energy more. Additionally, the standard deviations are small (relative to the mean), indicating high certainty in these predictions.

The results thus clearly show that with a high runtime, different endpoints consume significantly different amounts of energy, even though they were all evaluated at the same runtime. The preiously mentioned extremly high correlation between runtime and energy consumption, thus get a little smaller when we work with extreme runtime values.

Information Criteria Analysis

We want to analyze and compare the models used to evaluate H1, H2, and H3. We do this using **information criteria** based on the **Widely Applicable Information Criterion** (WAIC) and the **Leave-One-Out Information Criterion** (LOOIC). These criteria help us assess the goodness of fit of our models while penalizing for complexity.

We also intent to compare the three models, using the `az.compare` function, again measuring on both WAIC and LOOIC. This allows us to determine which model is best at *predicting energy consumption* for unseen data in this context.

Individual Model Analysis

In [36]:
```
# For H1 model
waic_h1 = az.waic(trace_h1)
loo_h1 = az.loo(trace_h1)
print("H1 Model WAIC:\n", waic_h1)
print("\nH1 Model LOO:\n", loo_h1)

# For H2 model
waic_h2 = az.waic(trace_h2)
loo_h2 = az.loo(trace_h2)
print("\nH2 Model WAIC:\n", waic_h2)
print("\nH2 Model LOO:\n", loo_h2)

# For H3 model
waic_h3 = az.waic(trace_h3)
```

```
loo_h3 = az.loo(trace_h3)
print("\nH3 Model WAIC:\n", waic_h3)
print("\nH3 Model LOO:\n", loo_h3)
```

```
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/arviz/stats/stats.py:1653: UserWarning: For one or more samples the posterior variance of the log predictive densities exceeds 0.4. This could be indication of WAIC starting to fail.
See http://arxiv.org/abs/1507.04544 for details
  warnings.warn(
```

```
H1 Model WAIC:
 Computed from 8000 posterior samples and 1960 observations log-likelihood matrix.

          Estimate       SE
elpd_waic  -189.54    61.34
p_waic       10.55        -

There has been a warning during the calculation. Please check the results.

H1 Model LOO:
 Computed from 8000 posterior samples and 1960 observations log-likelihood matrix.

         Estimate       SE
elpd_loo  -189.56    61.34
p_loo       10.56        -
------

Pareto k diagnostic values:
                         Count   Pct.
(-Inf, 0.70]   (good)     1960  100.0%
   (0.70, 1]   (bad)         0    0.0%
   (1, Inf)   (very bad)    0    0.0%
```

```
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/arviz/stats/stats.py:1653: UserWarning: For one or more samples the posterior variance of the log predictive densities exceeds 0.4. This could be indication of WAIC starting to fail.
See http://arxiv.org/abs/1507.04544 for details
  warnings.warn(
```

```
H2 Model WAIC:
 Computed from 8000 posterior samples and 1960 observations log-likelihood
matrix.

          Estimate       SE
elpd_waic  -188.54    60.43
p_waic       10.17        -

There has been a warning during the calculation. Please check the results.

H2 Model LOO:
 Computed from 8000 posterior samples and 1960 observations log-likelihood
matrix.

         Estimate       SE
elpd_loo  -188.55    60.43
p_loo       10.18        -
------

Pareto k diagnostic values:
                         Count   Pct.
(-Inf, 0.70]   (good)     1960  100.0%
   (0.70, 1]   (bad)         0    0.0%
   (1, Inf)   (very bad)    0    0.0%
```

```
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/arv
iz/stats/stats.py:1653: UserWarning: For one or more samples the posterior
variance of the log predictive densities exceeds 0.4. This could be indicat
ion of WAIC starting to fail.
See http://arxiv.org/abs/1507.04544 for details
  warnings.warn(
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/arv
iz/stats/stats.py:795: UserWarning: Estimated shape parameter of Pareto dis
tribution is greater than 0.70 for one or more samples. You should consider
using a more robust model, this is because importance sampling is less like
ly to work well if the marginal posterior and LOO posterior are very differ
ent. This is more likely to happen with a non-robust model and highly influ
ential observations.
  warnings.warn(
```

```
H3 Model WAIC:
 Computed from 8000 posterior samples and 1960 observations log-likelihood
matrix.

         Estimate       SE
elpd_waic  3871.62   216.68
p_waic       97.91        -

There has been a warning during the calculation. Please check the results.

H3 Model LOO:
 Computed from 8000 posterior samples and 1960 observations log-likelihood
matrix.

        Estimate       SE
elpd_loo  3876.33   212.79
p_loo       93.19        -

There has been a warning during the calculation. Please check the results.
------

Pareto k diagnostic values:
                         Count   Pct.
(-Inf, 0.70]   (good)     1957   99.8%
   (0.70, 1]   (bad)         2    0.1%
   (1, Inf)   (very bad)    1    0.1%
```

Interpretation of Information Criteria

- **H1:**

  The model used to test H1 has a WAIC estimate (`elpd_waic`) of -189.47 with 10.59 effective parameters (`p_waic`). However, we see that we get a warning stating "For one or more samples the posterior variance of the log predictive densities exceeds 0.4. This could be indication of WAIC starting to fail." This suggests that there exists potential unreliability due to high varaince in the posterior for some predictive densities.

  The model's LOOIC estimate (`elpd_loo`) is -189.48 with 10.61 effective parameters (`p_loo`). Importantly, the Pareto k diagnostics are all in the "good" range (k < 0.7) for all 1960 observatios, suggesting that the LOOIC estimate is reliable.

  Based on the good Pareto k diagnostics for LOOIC and the WAIC warning, we consider the LOOIC estimate to be more trustworthy to evaluate H1's out-of-sample predictive performance, however both criteria suggest similar levels of predictive performance and model complexity.

- **H2:**

  The H2 model's information criteria are similar to H1's. A WAIC estimate (`elpd_waic`) is -188.8, with an effective number of parameters (`p_waic`) of

10.49. Again, we get a warning about the posterior variance of the log predictive densities exceeding 0.4, indicating potential unreliability.

The LOOIC estimate (`elpd_loo`) is -188.81 for the H2 model with 10.5 effective parameters (`p_loo`). The Pareto k diagnostics are again all in the "good" range (k < 0.7) for all 1960 observations, showing reliable LOOIC estimates.

We again consider the LOOIC results more reliable for assessing H2's out-of-sample predictive performance, but both criteria suggest similar levels of predictive performance and model complexity.

- **H3:**

The H3 model produces a WAIC estimate (`elpd_waic`) of 3869.25, with a high effective number of parameters (`p_waic`) at 97.45. A warning about high posterior variance was once again shown, indicating potential unreliability.

The LOOIC estimate (`elpd_loo`) is 3877.72 with 88.99 effective parameters (`p_loo`). A warning was also shown for LOOIC, stating that "Estimated shape parameter of Pareto distribution is greater than 0.70 for one or more samples. You should consider using a more robust model, this is because importance sampling is less likely to work well if the marginal posterior and LOO posterior are very different. This is more likely to happen with a non-robust model and highly influential observations." This indiactes that the estimated Pareto k shape parameter exceeds 0.7 for some samples. This is seen in the Pareto k diagnostics, where 1 observation is classified as "bad" (0.7 < k <= 1), while 2 are "very bad" (k > 1). These few problematic k-values suggest that the LOOIC estimate might be less reliable because of influential observations.

Because of the warnings for both criteria, and the presence of high Pareto k values, we should be cautious when interpreting the information criteria for the H3 model. The model is more complex than those for H1 and H2, and thus its out-of-sample predictive performance may be less reliable, likely influenced by a few specific observations.

Model Comparison

In [37]:

```
# Dict of models
models_dict = {'H1': trace_h1, 'H2': trace_h2, 'H3': trace_h3}

# Comparison using LOO
comparison_loo = az.compare(models_dict)
print('\nModel Comparison (LOO):\n', comparison_loo)

# Comparison using WAIC
```

```
comparison_waic = az.compare(models_dict, ic = 'waic')
print('\nModel Comparison (WAIC):\n', comparison_waic)
```

```
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/arv
iz/stats/stats.py:795: UserWarning: Estimated shape parameter of Pareto dis
tribution is greater than 0.70 for one or more samples. You should consider
using a more robust model, this is because importance sampling is less like
ly to work well if the marginal posterior and LOO posterior are very differ
ent. This is more likely to happen with a non-robust model and highly influ
ential observations.
  warnings.warn(
Model Comparison (LOO):
     rank     elpd_loo      p_loo    elpd_diff    weight          se  \
H3     0  3876.333223  93.194694     0.000000  0.987254  212.794041
H2     1  -188.549738  10.179166  4064.882961  0.012746   60.430086
H1     2  -189.555236  10.556435  4065.888459  0.000000   61.337978

           dse  warning scale
H3    0.000000     True   log
H2  198.488818    False   log
H1  198.272030    False   log
```

```
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/arv
iz/stats/stats.py:1653: UserWarning: For one or more samples the posterior
variance of the log predictive densities exceeds 0.4. This could be indicat
ion of WAIC starting to fail.
See http://arxiv.org/abs/1507.04544 for details
  warnings.warn(
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/arv
iz/stats/stats.py:1653: UserWarning: For one or more samples the posterior
variance of the log predictive densities exceeds 0.4. This could be indicat
ion of WAIC starting to fail.
See http://arxiv.org/abs/1507.04544 for details
  warnings.warn(
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/arv
iz/stats/stats.py:1653: UserWarning: For one or more samples the posterior
variance of the log predictive densities exceeds 0.4. This could be indicat
ion of WAIC starting to fail.
See http://arxiv.org/abs/1507.04544 for details
  warnings.warn(
Model Comparison (WAIC):
     rank    elpd_waic     p_waic    elpd_diff    weight          se  \
H3     0  3871.617616  97.910301     0.000000  0.987252  216.679093
H2     1  -188.535615  10.165044  4060.153231  0.012748   60.427081
H1     2  -189.543837  10.545037  4061.161453  0.000000   61.335908

           dse  warning scale
H3    0.000000     True   log
H2  202.344125     True   log
H1  202.126182     True   log
```

Interpretation of Information Criteria Comparison

Based on both LOOIC and WAIC criteria, the model for H3 ranks as the best-fitting by a significant amount, as seen by the highest `elpd_loo` (3877.72) and `elpd_waic` (3869.25) values, and a weight of approximate 0.987 in both comparisons. As such, it

seems the H3 model has considerably better accuracy when predicting out-of-sample data than the models for H1 and H2.

The models for H2 and H1 rank second and third, respectively, although they have very similar `elpd` values (all around -189) and complexity (`p_loo` / `p_waic` around 10.5). The `elpd_diff` between these two models and the H3 models is very large (over 4000), supporting the indication of the H3 model having the best fit.

However, we should note:

- Model H3 is significantly more complex (for instance shown by its `p_loo` of 88.99, vs. approximately 10.5 for the H1/H2 models). This also makes intuitive sense, given the model used to test H3 is multilevel, while the H1 and H2 models are simple linear regression models.
- Both LOOIC and WAIC calculations generated warnings for H3, while H1 and H2 only saw warnings for WAIC. These warnings suggest that information criteria estimates might not be 100% reliable, particularly for H3.

We still strongly prefer H3 when comparing the models, due to its significantly larger ELPD values. And while warnings may mean the reliability of the estimates of predictive performance might not be perfect, they don't necessarily invalidate the model's superior ranking.

We should also keep in mind that we are comparing models with different explanatory variables and hypotheses, perhaps making it difficult to directly compare them. However, we can still conclude that for the goal of achieving the best *overall prediction* of energy consumption, a slightly more complex model like the one used for H3 is preferred in this context.

Ordinal Regression Model

We further want to explore the effect of `framework` on energy consumption by using an ordinal regression model. This model allows us to categorize energy consumption into discrete levels, defined as A (0-0.2 joules), B (0.2-0.4 joules), and C (>=0.4 joules). We use the same baseline framework as in H1, `c-sharp-razor`.

To do this, we first define the new ordinal model as such:

$$
\begin{aligned}
m_i &\sim \text{OrderedLogistic}(\eta_i, \boldsymbol{c}) && [\text{Likelihood for observed energy mark } m_i \\
\eta_i &= \alpha + \beta_{F[i]} && [\text{Linear model for latent variable } \eta_i \text{ (logi} \\
\alpha &\sim \mathcal{N}(0, 1.5) && [\text{Prior for intercept } \alpha \text{ (baseline framewo} \\
\beta_{F[i]} &\sim \mathcal{N}(0, 1.0) && [\text{Prior for framework effects } \beta_f \text{ (for each} \\
c_k &\sim \mathcal{N}(0, 1.5) && [\text{Priors for cutpoints } c_k, \text{ for } k = 1, \ldots, K \\
&\text{such that } c_1 < c_2 < \cdots < c_{K-2} && [\text{Cutpoints are ordered}]
\end{aligned}
$$

where:

- K is the number of energy marks (3 in this case: A, B, C).
- m_i is the observed energy mark (0 for mark A, 1 for mark B, 2 for mark C) for observation i, where $m_i \in \{0, 1, \ldots, K-1\}$.
- $F[i]$ is the index of the framework for observation i.
- η_i is the underlying continuous latent variable for observation i, modeled on the logit scale.
- $\boldsymbol{c}$ is a vector of $K-1$ cutpoints $(c_1, c_2, \ldots, c_{K-2})$ dividing the latent variable η_i into K ordered categories.
  - $P(m_i = 0) = \text{logistic}(\boldsymbol{c}_1 - \eta_i)$
  - $P(m_i = j) = \text{logistic}(\boldsymbol{c}_{j+1} - \eta_i) - \text{logistic}(\boldsymbol{c}_j - \eta_i)$ for $0 < j < K-1$
  - $P(m_i = K-1) = 1 - \text{logistic}(\boldsymbol{c}_{K-2} - \eta_i)$
- α is the intercept, representing the value of the latent variable η_i for the baseline framework `c-sharp-razor`.
- $\beta_{F[i]}$ is the effect of the frameework $F[i]$ on the latent variable η_i relative to the baseline.
- The priors for α, $\beta_{F[i]}$, and c_k are on the logit scale, which explains the different `mu` and `sigma` values compared to the previous model (which measured on a joules scale). The values `Normal(0, 1.5)` and `Normal(0, 1.0)` are common weakly informative priors for parameters on the logit scale.

Data Preparation

We first need to create the ordinal variable `energy_mark` based on the energy consumption values. We define the cutoffs for the three categories (A, B, C) and then create the new variable.

In [38]:

```
# Define the cut-points for energy marks
cutoffs = [0.0, 0.2, 0.4, float('inf')] # 0.0-0.2 (A), 0.2-0.4
(B), >=0.4 (C)
labels = [0, 1, 2] # Marks A, B, C

# Create the 'energy_mark' column in your dataframe df
df['energy_mark'] = pd.cut(df['energy_consumption'], bins =
cutoffs, labels = labels, right = False, include_lowest = True)
```

```
df['energy_mark'] = df['energy_mark'].astype(int)

# Number of energy mark categories
K = len(labels)
```

Prior Predictive Checks

We naturally want to check predictions from this ordinal model's priors as well. We do this similarly to the previous models, by sampling from the priors and plotting the resulting distributions.

In [39]:

```
# Define K
K = 3 # Number of categories (A, B, C)

# Define the model for prior predictive checks for the Ordinal
Model
with pm.Model() as model_ordinal_prior_pred:
    # Priors (on logit scale)
    alpha = pm.Normal('alpha', mu = 0, sigma = 1.5)
# Intercept for baseline framework
    betas = pm.Normal('betas', mu = 0, sigma = 1.0, shape =
X_h1.shape[1]) # Effects of other frameworks

    # Cutpoints
    cutpoints = pm.Normal('cutpoints', mu = 0, sigma = 1.5, shape
= K - 1, # K-1 cutpoints for K categories,
                          transform =
pm.distributions.transforms.ordered)

    # Linear model for the latent variable eta (logit scale)
    eta_values = pm.Deterministic('eta_values', alpha +
pm.math.dot(X_h1.values, betas))

    # Likelihood for prior predictive sampling
    m_simulated = pm.OrderedLogistic('m_simulated', eta =
eta_values, cutpoints = cutpoints, shape = df.shape[0])

    # Sample from the prior predictive distribution
    prior_pred_samples_ordinal =
pm.sample_prior_predictive(samples = 500, random_seed = 42)
```

```
# Flatten the 2D array of simulated marks to 1D
# # Plot the distribution
# az.plot_dist(simulated_m, kind='hist', hist_kwargs={'alpha':
0.8, 'bins': np.arange(-0.5, K + 0.5, 1), 'align': 'mid',
'rwidth': 0.8})
# plt.xlabel('Simulated Energy Mark (0=A, 1=B, 2=C)')
# plt.ylabel('Frequency / Density')
# plt.title('Prior Predictive Check: Distribution of Simulated
Energy Marks')
# plt.xticks(ticks=np.arange(K), labels=[f'Mark {i}' for i in
range(K)])
# plt.tight_layout()
# plt.show()
# Plot the prior predictive distribution for simulated energy
marks (m_simulated)
print('Plotting Prior Predictive Distribution for m_simulated
(Energy Marks):')
simulated_m =
prior_pred_samples_ordinal.prior['m_simulated'].stack(samples=
('chain', 'draw')).values.flatten()


# simulated_m =
prior_pred_samples_ordinal.prior['m_simulated'].stack(samples =
('chain', 'draw')).values
az.plot_dist(simulated_m, kind = 'hist', hist_kwargs = {'alpha':
.8, 'bins': np.arange(-0.5, K + 0.5, 1), 'align':'mid',
'rwidth':0.8})
plt.xlabel('Simulated Energy Mark (0=A, 1=B, 2=C)')
plt.ylabel('Frequency / Density') # Arviz might plot density or
counts
plt.title('Prior Predictive Check: Distribution of Simulated
Energy Marks')
plt.xticks(ticks=np.arange(K), labels=[f'Mark {i}' for i in
range(K)])
plt.tight_layout()
plt.show()


# Plotting the prior predictive distribution for eta_values
(latent variable)
```

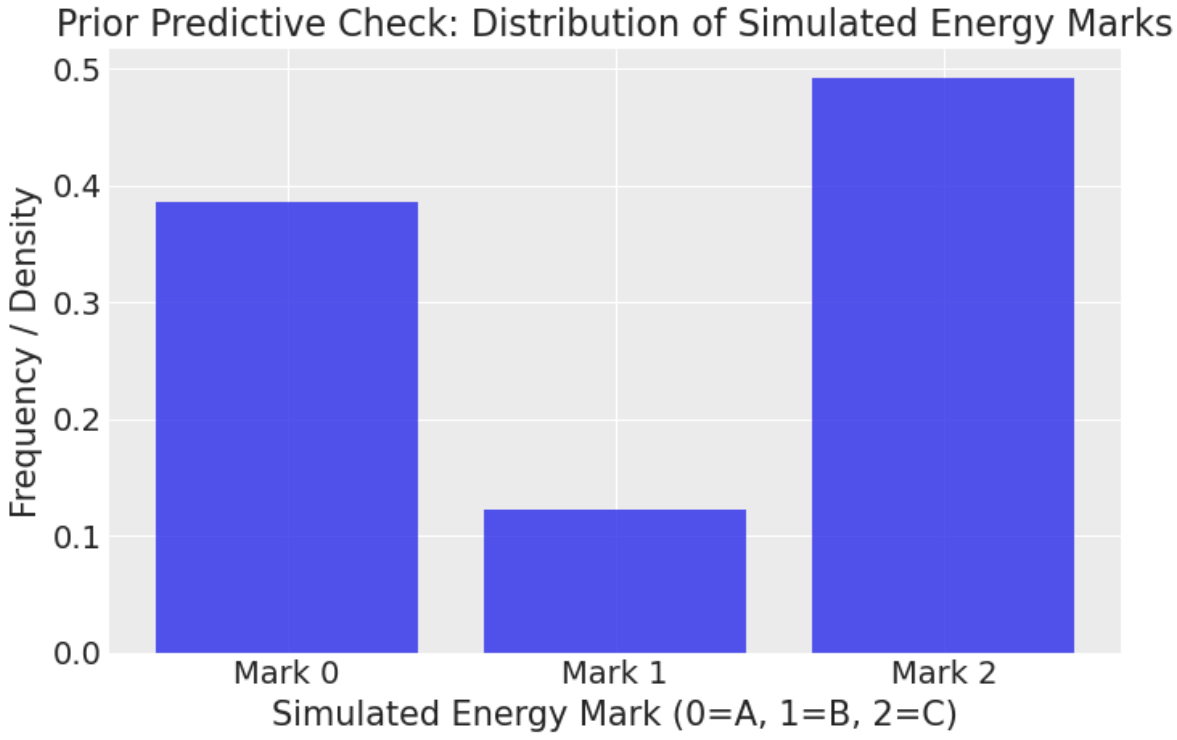
```
print('\nPlotting Prior Predictive Distribution for eta_values
(Latent Variable):')
eta_plot_values =
prior_pred_samples_ordinal.prior['eta_values'].stack(samples=
('chain', 'draw')).values.flatten()

# eta_plot_values =
prior_pred_samples_ordinal.prior['eta_values'].stack(samples =
('chain', 'draw'))
az.plot_dist(eta_plot_values, kind = 'hist', hist_kwargs =
{'alpha': .8, 'bins': 50})
plt.xlabel('Simulated Latent Variable (eta) on logit scale')
plt.ylabel('Density')
plt.title('Prior Predictive Check: Distribution of Simulated
Latent Variable (eta)')
x_min_eta, x_max_eta = np.min(eta_plot_values),
np.max(eta_plot_values)
x_ticks_eta = np.linspace(x_min_eta, x_max_eta, num = 15)
plt.xticks(x_ticks_eta, rotation = 45)
plt.tight_layout()
plt.show()

# Plotting the distributions of the priors themselves for alpha,
betas, and cutpoints
print('\nPlotting Prior Distributions for Parameters (logit
scale):')
fig, axes = plt.subplots(1, 3, figsize = (18, 5))

# Alpha
alpha_prior_vals =
prior_pred_samples_ordinal.prior['alpha'].stack(samples=('chain',
'draw')).values.flatten()
# Betas
betas_prior_vals =
prior_pred_samples_ordinal.prior['betas'].stack(samples=('chain',
'draw')).values.flatten()
# Cutpoints
cutpoints_prior_vals =
prior_pred_samples_ordinal.prior['cutpoints'].stack(samples=
('chain', 'draw')).values.flatten()

# Alpha
```

```
# alpha_prior_vals =
prior_pred_samples_ordinal.prior['alpha'].stack(samples=('chain',
'draw'))
az.plot_dist(alpha_prior_vals, ax = axes[0], label = 'alpha',
kind = 'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks_alpha = np.linspace(np.min(alpha_prior_vals),
np.max(alpha_prior_vals), num = 10)
axes[0].set_xticks(x_ticks_alpha)
axes[0].set_xticklabels([f'{tick:.1f}' for tick in
x_ticks_alpha], rotation = 45)
axes[0].set_title('Prior Distribution for alpha (Baseline
effect)')
axes[0].set_xlabel('Value (logit scale)')
axes[0].set_ylabel('Density')

# Betas
# betas_prior_vals =
prior_pred_samples_ordinal.prior['betas'].stack(samples=('chain',
'draw'))
az.plot_dist(betas_prior_vals, ax = axes[1], label = 'betas',
kind = 'hist', hist_kwargs = {'alpha': .8, 'bins': 30})
x_ticks_betas = np.linspace(np.min(betas_prior_vals),
np.max(betas_prior_vals), num = 10)
axes[1].set_xticks(x_ticks_betas)
axes[1].set_xticklabels([f'{tick:.1f}' for tick in
x_ticks_betas], rotation = 45)
axes[1].set_title('Prior Distribution for betas (Framework
effects)')
axes[1].set_xlabel('Value (logit scale)')
axes[1].set_ylabel('Density')

# Cutpoints
# cutpoints_prior_vals =
prior_pred_samples_ordinal.prior['cutpoints'].stack(samples =
('chain', 'draw'))
az.plot_dist(cutpoints_prior_vals, ax = axes[2], label =
'cutpoints', kind = 'hist', hist_kwargs = {'alpha': .8, 'bins':
30})
x_ticks_cuts = np.linspace(np.min(cutpoints_prior_vals),
np.max(cutpoints_prior_vals), num = 10)
axes[2].set_xticks(x_ticks_cuts)
axes[2].set_xticklabels([f'{tick:.1f}' for tick in x_ticks_cuts],
```

```
rotation = 45)
axes[2].set_title('Prior Distribution for cutpoints')
axes[2].set_xlabel('Value (logit scale)')
axes[2].set_ylabel('Density')

plt.tight_layout()
plt.show()

# Range the range or typical values of eta and cutpoints
print('\nSummary of prior predictive samples for eta_values:')
# print(pd.Series(eta_plot_values.values).describe())
print(pd.Series(eta_plot_values).describe())

print('\nSummary of prior predictive samples for cutpoints
(flattened):')
print(pd.Series(cutpoints_prior_vals.flatten()).describe())

print('\nExample of simulated energy marks distribution (first
500 samples):')
print(pd.Series(simulated_m).value_counts(normalize=True).sort_inde
```

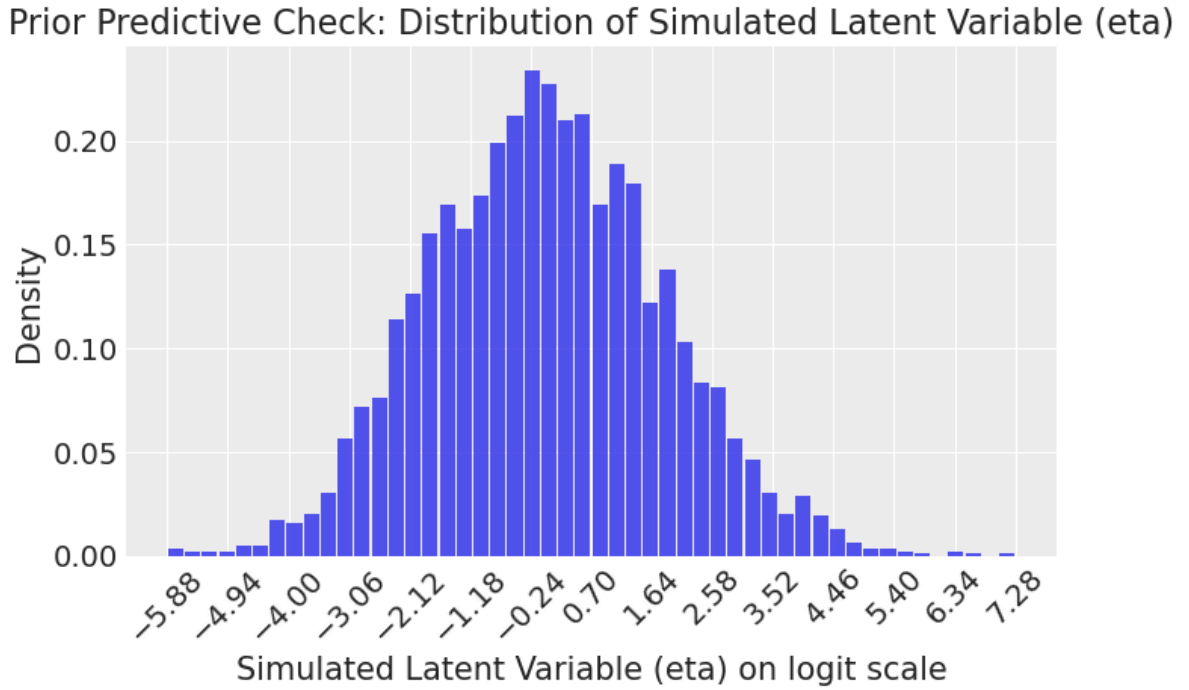
```
Sampling: [alpha, betas, cutpoints, m_simulated]
Plotting Prior Predictive Distribution for m_simulated (Energy Marks):
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/202244677
7.py:44: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



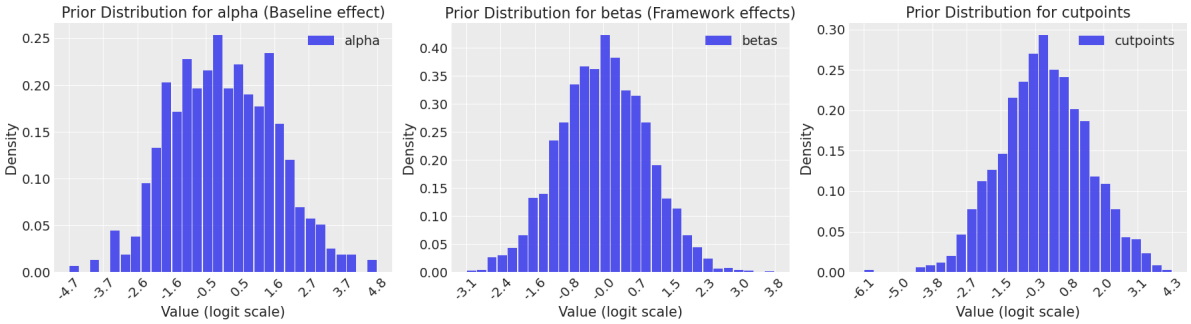

```
Plotting Prior Predictive Distribution for eta_values (Latent Variable):
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/2022446777.py:60: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```




```
Plotting Prior Distributions for Parameters (logit scale):
```

```
/var/folders/yw/s572_ycn0n5d19g5p1z13l3c0000gn/T/ipykernel_13428/2022446777.py:104: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



```
Summary of prior predictive samples for eta_values:
count    980000.000000
mean          0.006703
std           1.806375
min          -5.875902
25%          -1.276607
50%          -0.028709
75%           1.220234
max           7.279578
dtype: float64

Summary of prior predictive samples for cutpoints (flattened):
count    1000.000000
mean       -0.016129
std         1.469381
min        -6.139331
25%        -1.005575
50%        -0.037568
75%         0.972664
max         4.306813
dtype: float64

Example of simulated energy marks distribution (first 500 samples):
0    0.385518
1    0.122658
2    0.491823
Name: proportion, dtype: float64
```

Interpretation of Prior Predictive Checks for Ordinal Model

1. **Distribution of Simulated Energy Marks (`m_simulated`):**
    - The priors generate observations across all three energy mark categories (0, 1, 2).
    - The sample distribution (mark 0: approx. 38.55%, mark 1: approx. 12.27%, mark 2: approx. 49.18%) shows that priors do not overly favor any specific category, though there's a higher chance of predicting marks 0 or 2. The distribution seems reasonable for the weakly informed prior, as the data can be allowed to inform the actual probabilities for each energy mark.
2. **Distribution of Simulated Latent Variable (`eta_values`):**
    - The logit scale latent variable η is centered around 0 (mean: approx. 0.007) with a standard deviation of approximately 1.81.
    - The range (approx. -5.9 to 7.3) suggests that the priors allow η to vary across a fairly plausible range of logit values, in turn influencing the porbabilities of the different energy marks. This spread is also consistent with the `Normal(0, 1.5)` prior for `alpha` and `Normal(0, 1.0)` for `betas`.
3. **Distribution of Prior Parameters (`alpha`, `beta`, `c`):**
    - The prior distribution plots seem to follow the specified prior distribution parameters well.

- The `cutpoints` summary (mean: approx. -0.016, std: approx. 1.47) fits with its `Normal(0, 1.5)` prior, and the the ordered transform ensures $c_1 < c_2$.

Model Fitting and Sampling

We fit the model with `PyMC` and sample from the posterior distribution using the No-U-Turn Sampler (NUTS). The model is specified in a `with` block, and we use `pm.sample()` to draw samples from the posterior distribution. We set the number of samples to 2000 and the number of tuning steps to 1000. We use `chains=4` to run four chains in parallel for better convergence diagnostics.

Furthermore, a visualization of the model setup is printed through `pm.model_to_graphviz`.

In [40]:
```
# Define y as the energy marks
y_marks = df['energy_mark'].values

with pm.Model() as model_ordinal:
    # Priors (on logit scale)
    alpha = pm.Normal('alpha', mu = 0, sigma = 1.5)
# Intercept for baseline framework (c-sharp-razor)
    betas = pm.Normal('betas', mu = 0, sigma = 1.0, shape =
X_h1.shape[1]) # Effects of other frameworks

    # Cutpoints
    cutpoints = pm.Normal('cutpoints', mu = 0, sigma = 1.5, shape
= K - 1,

transform=pm.distributions.transforms.ordered)

    # Linear model for the latent variable eta (logit scale)
    eta = alpha + pm.math.dot(X_h1.values, betas)

    # Likelihood
    m_obs = pm.OrderedLogistic('m_obs', eta = eta, cutpoints =
cutpoints, observed = y_marks)

    # Sampling from the posterior
    trace_ordinal = pm.sample(2000, tune = 1000, target_accept =
.95, return_inferencedata = True, random_seed = 42, idata_kwargs
= {'log_likelihood': True})
```

```
# Visualize the model structure
pm.model_to_graphviz(model_ordinal)
```

```
Initializing NUTS using jitter+adapt_diag...
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/pytensor/tensor/elemwise.py:735: RuntimeWarning: divide by zero encountered in log
  variables = ufunc(*ufunc_args, **ufunc_kwargs)
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/pytensor/tensor/elemwise.py:735: RuntimeWarning: divide by zero encountered in log
  variables = ufunc(*ufunc_args, **ufunc_kwargs)
/Users/chris/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/pytensor/compile/function/types.py:992: RuntimeWarning: invalid value encountered in accumulate
  outputs = vm() if output_subset is None else vm(output_subset=output_subset)
```

```
---------------------------------------------------------------------------
SamplingError                             Traceback (most recent call last)
Cell In[40], line 20
     17     m_obs = pm.OrderedLogistic('m_obs', eta = eta, cutpoints = cutp
oints, observed = y_marks)
     19     # Sampling from the posterior
---> 20     trace_ordinal = pm.sample(2000, tune = 1000, target_accept = .9
5, return_inferencedata = True, random_seed = 42, idata_kwargs = {'log_like
lihood': True})
     22 # Visualize the model structure
     23 pm.model_to_graphviz(model_ordinal)

File ~/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/pymc/samp
ling/mcmc.py:804, in sample(draws, tune, chains, cores, random_seed, progre
ssbar, progressbar_theme, step, var_names, nuts_sampler, initvals, init, ji
tter_max_retries, n_init, trace, discard_tuned_samples, compute_convergence
_checks, keep_warning_stat, return_inferencedata, idata_kwargs, nuts_sample
r_kwargs, callback, mp_ctx, blas_cores, model, compile_kwargs, **kwargs)
    802         [kwargs.setdefault(k, v) for k, v in nuts_kwargs.items()]
    803     with joined_blas_limiter():
--> 804         initial_points, step = init_nuts(
    805             init=init,
    806             chains=chains,
    807             n_init=n_init,
    808             model=model,
    809             random_seed=random_seed_list,
    810             progressbar=progressbar,
    811             jitter_max_retries=jitter_max_retries,
    812             tune=tune,
    813             initvals=initvals,
    814             compile_kwargs=compile_kwargs,
    815             **kwargs,
    816         )
    817 else:
    818     # Get initial points
    819     ipfns = make_initial_point_fns_per_chain(
    820         model=model,
    821         overrides=initvals,
    822         jitter_rvs=set(),
    823         chains=chains,
    824     )

File ~/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/pymc/samp
ling/mcmc.py:1504, in init_nuts(init, chains, n_init, model, random_seed, p
rogressbar, jitter_max_retries, tune, initvals, compile_kwargs, **kwargs)
   1502 logp_dlogp_func = model.logp_dlogp_function(ravel_inputs=True, **co
mpile_kwargs)
   1503 logp_dlogp_func.trust_input = True
-> 1504 initial_points = _init_jitter(
   1505     model,
   1506     initvals,
   1507     seeds=random_seed_list,
   1508     jitter="jitter" in init,
   1509     jitter_max_retries=jitter_max_retries,
   1510     logp_dlogp_func=logp_dlogp_func,
   1511 )
```

```
   1513 apoints = [DictToArrayBijection.map(point) for point in initial_poi
nts]
   1514 apoints_data = [apoint.data for apoint in apoints]

File ~/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/pymc/samp
ling/mcmc.py:1390, in _init_jitter(model, initvals, seeds, jitter, jitter_m
ax_retries, logp_dlogp_func)
   1387 if not np.isfinite(point_logp):
   1388     if i == jitter_max_retries:
   1389         # Print informative message on last attempted point
-> 1390         model.check_start_vals(point)
   1391     # Retry with a new seed
   1392     seed = rng.integers(2**30, dtype=np.int64)

File ~/opt/anaconda3/envs/prpro-2025/lib/python3.12/site-packages/pymc/mode
l/core.py:1769, in Model.check_start_vals(self, start, **kwargs)
   1766 initial_eval = self.point_logps(point=elem, **kwargs)
   1768 if not all(np.isfinite(v) for v in initial_eval.values()):
-> 1769     raise SamplingError(
   1770         "Initial evaluation of model at starting point failed!\n"
   1771         f"Starting values:\n{elem}\n\n"
   1772         f"Logp initial evaluation results:\n{initial_eval}\n"
   1773         "You can call `model.debug()` for more details."
   1774     )

SamplingError: Initial evaluation of model at starting point failed!
Starting values:
{'alpha': array(-0.31962516), 'betas': array([ 0.33575501, -0.96307366, -0.
18452522,  0.51307857,  0.36098139,
       -0.72161403]), 'cutpoints_ordered__': array([0.84856694,       -in
f])}

Logp initial evaluation results:
{'alpha': -1.35, 'betas': -6.51, 'cutpoints': -inf, 'm_obs': -inf}
You can call `model.debug()` for more details.
```

We see that the above cell returns a `SamplingError` when attempting to sample from the posterior distribution after fitting the model.

While we would naturally like to fix this error to sample from the ordinal model, we find that we unfortunately do not have the time to do so. However, we still believe that the model setup is correct, and would have liked to explore this direction for predicting energy consumption further.

Overall Conclusion

In this notebook report, we have analysed the energy consumption and runtimes of different web frameworks and programming languages using Bayesian regression models. We explored three hypotheses (H1, H2, and H3) to investigate the impact of framework, language, and runtime on energy consumption.

Our analyses strongly support all three hypotheses, with the following conclusions:

- **H1**: The web framework used likely has a significant impact on energy consumption, with `c-sharp-razor` consuming the most energy.
- **H2**: The programming language used likely has a significant impact on energy consumption, with `javascript` being the most energy efficient.
- **H3**: Runtime has a stronger impact on energy consumption for some API endpoints than others.

Addtionally, we have performed prior and posterior predictive checks, model diagnostics, and information criteria analysis (WAIC and LOOIC) to assess the fit of each model used to analyse H1, H2, and H3, as well as to compare their predictive performance. We used a multilevel model to analyse H3, capturing endpoint-specific effects of runtime on energy consumption. Overall, we find strong evidence for the influence of framwork, language, and runtime on the energy consumption.