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

Probabilistic Programming 2025 Exam

Students: Oleksandr Adamov (olea@itu.dk) Costel Gutu (cogu@itu.dk) Vladislav Konjushenko (vlko@itu.dk)

Introduction

This report details the analysis of energy consumption across different implementations of a web application, undertaken as part of the Probabilistic Programming 2025 Exam. The objective is to determine whether significant differences in energy consumption exist based on the programming language, web framework, or specific API endpoints used.

Project Goals

The goals of this analysis are:

1. To investigate the impact of **web framework**, **programming language**, and **API endpoint dynamics (specifically the interaction between runtime and endpoint)** on the energy consumption of a web application by analyzing the provided `dataset.csv`.
2. To construct, fit, and meticulously validate a series of **Bayesian regression models** using PyMC, each tailored to address one of the specific hypotheses (H1, H2, H3). Validation will encompass **prior predictive checks, posterior predictive checks, and a thorough assessment of MCMC sampling quality** (e.g., visual inspection of trace plots, $\hat{R}$ statistics, and Effective Sample Sizes).
3. To test the three hypotheses:
    - **H1:** The `c-sharp-razor` web framework 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.

Conclusions will be drawn based on the analysis of posterior distributions derived from the models.
4. Performing **counterfactual analysis** (especially in the context of H3), to better understand energy consumption patterns and inform potential optimizations and energy-conscious choices in web application development.

Methodology Overview

The core of this project lies in the application of Bayesian statistical modeling. This involves:

1. **Data Exploration and Preprocessing:** Loading, examining, and preparing the `dataset.csv` data, including extracting programming languages from the `Application` field and any necessary transformations.
2. **Prior Elicitation:** Defining prior beliefs about model parameters, using informed or uninformative priors as appropriate, and justifying these choices.
3. **Model Building:** Constructing Bayesian regression models using PyMC to address the stated hypotheses. This includes careful consideration of model structure to accurately represent the relationships under investigation.
4. **Model Checking and Refinement:** Assessing model fit and sampling quality (e.g., through trace plots, summary statistics, $\hat{R}$, ESS, and predictive checks) and iteratively improving models if necessary.
5. **Inference and Hypothesis Testing:** Drawing conclusions from the posterior distributions of model parameters to evaluate the hypotheses, and clearly articulating the evidence for or against each.

Key Hypotheses

Throughout this project, we will investigate the following hypotheses as specified in the exam description:

- **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.

```
In [1]: #Importing necessary libraries
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        import pymc as pm
        import arviz as az
```

Analysis of Hypothesis H1: Energy Consumption of `c-sharp-razor`

1. Introduction

This section details the investigation of Hypothesis H1: **The web framework `c-sharp-razor` consumes more energy than any other web framework in the dataset.**

To address this hypothesis, we will perform an exploratory data analysis to understand the distribution of energy consumption across different frameworks, apply necessary data transformations, and then build a Bayesian statistical model using PyMC. The model will estimate the mean energy consumption for each framework, allowing us to compare `c-sharp-razor` (referred to as `razor` in the framework data) with others and assess the credibility of H1.

2. Data Loading and Preparation

The analysis begins by loading the `dataset.csv`. The `application` column contains combined information about the programming language and the web framework. For H1, we are interested in the web framework.

Key preparation steps include:

1. **Loading Data**: The dataset is loaded into a pandas DataFrame.
2. **Feature Extraction**: The `application` column is split to create separate `Language` and `Framework` columns.
3. **Categorical Encoding**: The `Framework` column (text-based) is converted into numerical indices (`framework_idx`) for use in the statistical model. A mapping (`framework_map`) is retained to link these indices back to the framework names.

The following code performs these initial steps.

```
In [2]: # 1. Load Data
        df = pd.read_csv('dataset.csv') # Assuming header is correctly infe
```

```
# 2. Feature Extraction
# Splitting 'application' column (e.g., 'c-sharp-razor') into 'Lang
df[['Language', 'Framework']] = df['application'].str.rsplit('-', n

# 3. Categorical Encoding for Framework
df['framework_idx'] = df['Framework'].astype('category').cat.codes
framework_map = (
    df[['Framework', 'framework_idx']]
      .drop_duplicates()
      .sort_values('framework_idx')
      .reset_index(drop=True)
)
framework_names = framework_map['Framework'].tolist() # For plottin

print("Framework to Index Mapping:")
print(framework_map.to_string())
print(f"\nNumber of unique frameworks (J): {df['framework_idx'].nun
```

```
Framework to Index Mapping:
  Framework  framework_idx
0     actix              0
1   express              1
2     flask              2
3       gin              3
4   gorilla              4
5     razor              5
6   sinatra              6

Number of unique frameworks (J): 7
```

3. Exploratory Data Analysis (EDA) and Data Transformation

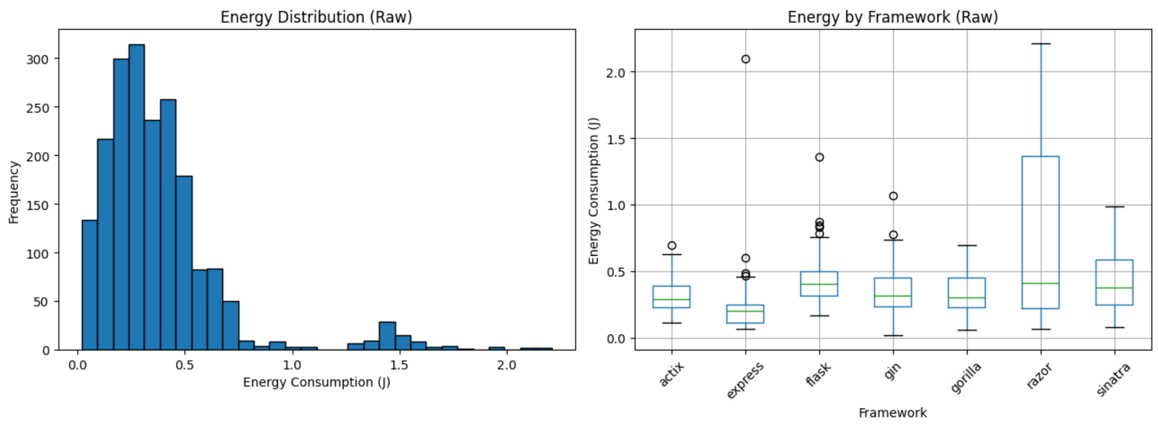
Before modeling, we explore the `energy_consumption` data to understand its distribution and identify any characteristics that might influence model choice, such as skewness or outliers.

3.1. Initial Distribution of Energy Consumption

A histogram of the raw `energy_consumption` and box plots grouped by `Framework` are generated.

In [3]:
```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

axes[0].hist(df['energy_consumption'], bins=30, edgecolor='k')
axes[0].set_title('Energy Distribution (Raw)')
axes[0].set_xlabel('Energy Consumption (J)')
axes[0].set_ylabel('Frequency')

df.boxplot(column='energy_consumption', by='Framework', rot=45, ax=
axes[1].set_title('Energy by Framework (Raw)')
axes[1].set_xlabel('Framework')
axes[1].set_ylabel('Energy Consumption (J)')
```

```
plt.suptitle('') # Suppress default suptitle if any
plt.tight_layout()
plt.show()

raw_skewness = df['energy_consumption'].skew()
print(f"Skewness of raw energy_consumption: {raw_skewness:.4f}")
```



```
Skewness of raw energy_consumption: 2.5745
```

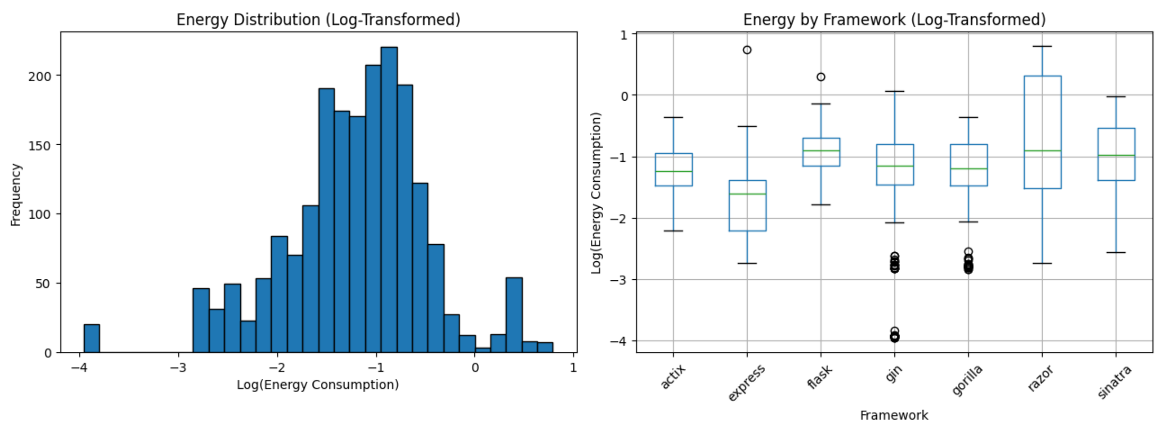
Skewness Analysis (Raw Data): The initial histogram shows a distinct right tail, indicating rare but very large energy consumption values (>1 J), while most measurements are concentrated in the 0.1-0.6 J range. The calculated skewness is approximately **2.5745**, confirming a strong positive skew. Such a distribution is not well-suited for models assuming normally distributed errors.

3.2. Log Transformation

To address the right skew and compress large outlier values, a log transformation is applied to `energy_consumption`. A small constant (1e-6) is added before taking the log to avoid issues with zero or near-zero values. The new variable is `logEnergy`.

In [4]:
```
df['logEnergy'] = np.log(df['energy_consumption'] + 1e-6)

fig, axes = plt.subplots(1, 2, figsize=(13, 5))

axes[0].hist(df['logEnergy'], bins=30, edgecolor='k')
axes[0].set_title('Energy Distribution (Log-Transformed)')
axes[0].set_xlabel('Log(Energy Consumption)')
axes[0].set_ylabel('Frequency')

df.boxplot(column='logEnergy', by='Framework', rot=45, ax=axes[1])
axes[1].set_title('Energy by Framework (Log-Transformed)')
axes[1].set_xlabel('Framework')
axes[1].set_ylabel('Log(Energy Consumption)')

plt.suptitle('')
plt.tight_layout()
plt.show()
```

```
log_skewness = df['logEnergy'].skew()
print(f"Skewness of logEnergy: {log_skewness:.4f}")
```



```
Skewness of logEnergy: -0.4899
```

Analysis of Log-Transformed Data: The histogram of `logEnergy` is much closer to a normal distribution, though with a slight left skew and moderate tails. The skewness value is significantly reduced. The box plots (both raw and log-transformed) indicate that `razor` and `flask` frameworks tend to have larger interquartile ranges. `gin` and `gorilla` show some strong low-value outliers in the log scale. The log transformation makes the data more amenable to modeling with a likelihood that assumes (or is robust to deviations from) normality, such as the Normal or Student-T distribution.

3.3. Standardization of the Response Variable

To unify the scale of the (log-transformed) energy consumption and allow for the use of more general, scale-independent priors in our Bayesian model, we standardize `logEnergy` to have a mean of 0 and a standard deviation of 1. This transformed variable is named `energy_z`.

Model coefficients for `energy_z` will be in "standard deviation" units, simplifying interpretation (e.g., how many standard deviations from the overall mean a particular framework's energy consumption lies).

In [5]:
```
e_mean, e_sd = df['logEnergy'].mean(), df['logEnergy'].std()
df['energy_z'] = (df['logEnergy'] - e_mean) / e_sd

print("Descriptive statistics for 'energy_z' by Framework (sorted b
stats_by_framework = (df
 .groupby('Framework')['energy_z']
 .agg(['mean','median','std', 'min', 'max', 'count'])
 .sort_values('mean', ascending=False)
)
print(stats_by_framework)
```

```
Descriptive statistics for 'energy_z' by Framework (sorted by mean):
               mean    median       std       min       max  count
Framework
razor      0.592130  0.430164  1.309890 -2.105408  2.751001    280
flask      0.337858  0.414704  0.566857 -0.780794  2.084228    280
sinatra    0.202525  0.328019  0.881967 -1.849344  1.639730    280
gorilla   -0.077961  0.014774  0.818815 -2.239366  1.167577    280
actix     -0.098035 -0.043867  0.649478 -1.362320  1.163356    280
gin       -0.245952  0.084741  1.214580 -3.765673  1.752087    280
express   -0.710565 -0.548494  0.740443 -2.098138  2.680258    280
```

Takeaways from Standardized Data Statistics:

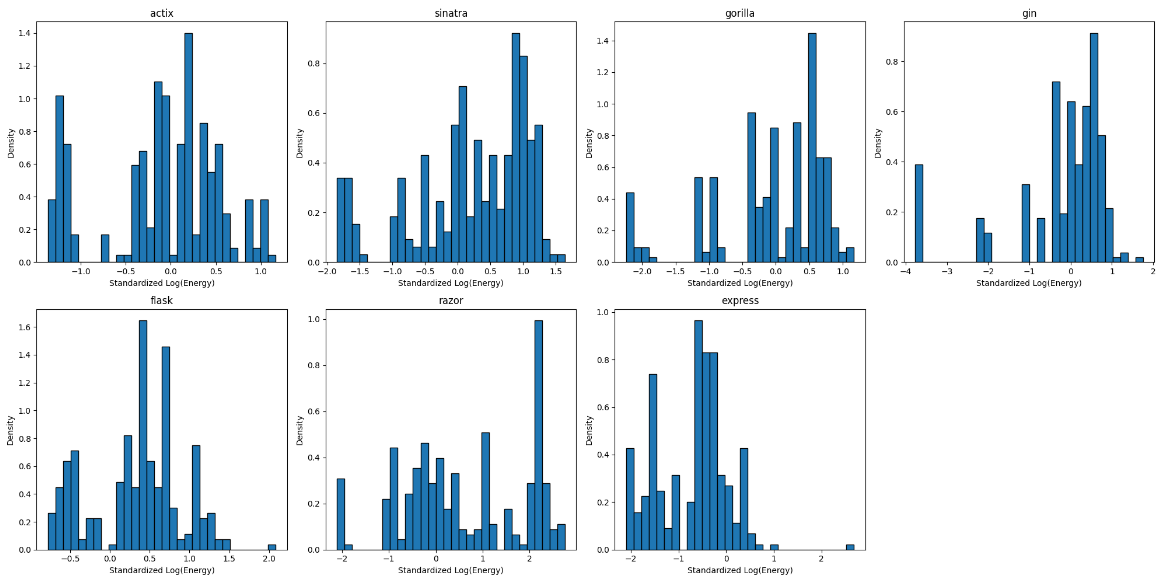
- **Means**: `razor` (mean `energy_z` ≈ 0.592) has the highest average standardized log energy consumption, providing initial, descriptive support for H1. `express` (mean `energy_z` ≈ -0.711) has the lowest.
- **Medians vs. Means**: For `razor`, the mean is greater than the median, suggesting a right tail (rare "expensive" runs). For `express`, the mean is less than the median, suggesting a left tail (rare "super-efficient" runs).
- **Standard Deviation**: `razor` (std ≈ 1.31) shows large within-framework variability. `flask` (std ≈ 0.57) shows more stable results. The high variability in some frameworks like `razor` and `gin` (std ≈ 1.21) suggests that a robust likelihood distribution, such as the Student-T distribution which has "heavy tails," would be appropriate for the model.
- **Outliers**: Wide min/max ranges (e.g., for `gin` and `razor`) further point to the presence of outliers.
- **Balanced Design**: Each framework has 280 observations, indicating a balanced design.

3.4. Distribution of `energy_z` per Framework

Histograms for `energy_z` for each framework are plotted to visually inspect their individual distributions and inform model choices, particularly regarding the likelihood and prior specifications.

In [6]:
```
framework_list = df['Framework'].unique() # Or use framework_names
fig, axes = plt.subplots(2, 4, figsize=(20, 10)) # Adjust grid size
axes = axes.flatten()

for i, fw_name in enumerate(framework_list):
    if i < len(axes): # Ensure we don't try to plot on non-existent
        ax = axes[i]
        values = df[df['Framework'] == fw_name]['energy_z']
        ax.hist(values, bins=30, edgecolor='k', density=True) # den
        ax.set_title(fw_name)
        ax.set_xlabel('Standardized Log(Energy)')
        ax.set_ylabel('Density')

# Turn off any unused axes if the number of frameworks is not a mul
```

```
for j in range(i + 1, len(axes)):
    axes[j].axis('off')

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



Model Implications from Histograms: The individual histograms reinforce the earlier observations:

- `actix` and `flask` appear relatively symmetric and close to normal.
- `express` and `gin` show pronounced left tails (very low energy runs).
- `razor` exhibits a strong right tail (high energy runs) and the largest dispersion.
- `gorilla` and `sinatra` also show some skewness and outliers.

These characteristics, especially the heavy tails and outliers for several frameworks, strongly motivate the use of a **Student-T distribution for the likelihood** in our Bayesian model. The Student-T distribution can better accommodate outliers compared to a Normal distribution, preventing them from unduly influencing the parameter estimates.

4. Bayesian Model Design for H1

To test H1, we model the standardized log energy consumption (`energy_z`) as a function of the web framework. The model aims to estimate a mean `energy_z` for each framework.

4.1. Model Specification

Let y_i be the `energy_z` for observation i, and fw_i be the framework index for that observation. The model is:

$$y_i \sim \text{StudentT}(\mu_i, \sigma, \nu)$$

$$\mu_i = \alpha + \beta_{fw_i}$$

Where:

- α: A global intercept, representing an overall mean level of `energy_z`.
- β_j: A coefficient for each framework j. In this parameterization (known as cell means when priors are set appropriately, or more typically as effects relative to a baseline if using sum-to-zero constraints or a reference category, which is not explicitly done here through prior construction but the priors are distinct), $\alpha + \beta_j$ effectively represents the mean `energy_z` for framework j. We will interpret β_j as the mean deviation for framework j from the global intercept α.
- σ: The scale parameter (standard deviation) of the Student-T distribution.
- ν: The degrees of freedom parameter for the Student-T distribution. Lower values of ν imply heavier tails (more robust to outliers). We add 1 to its prior to ensure $\nu > 1$.

4.2. Prior Specification

Priors are chosen to be weakly informative, guided by the exploratory data analysis:

- α **(Global Intercept)**:
  - $\alpha \sim \text{Normal}(\mu_\alpha, \sigma_\alpha)$
  - μ_α and σ_α are derived from the 2.5th and 97.5th percentiles of the overall `energy_z` distribution, aiming to cover the bulk of the data.
- β_j **(Framework-specific effects)**:
  - $\beta_j \sim \text{Normal}(\mu_{\beta_j}, \sigma_{\beta_j})$ for each framework j.
  - Unlike a fully hierarchical model where β_j would be drawn from a common distribution, here each β_j has its own prior mean (μ_{β_j}) and standard deviation (σ_{β_j}). These are also estimated using robust percentiles of `energy_z` *within each framework*. This approach gives each framework its own prior expectation and scale, which is deemed important for estimating the effect of `razor` distinctly.
- σ **(Scale parameter)**:
  - $\sigma \sim \text{HalfNormal}(\sigma_{scale})$
  - σ_{scale} is set using the 95th percentile of the absolute values of `energy_z` (approx. 2.167), expecting most residuals to fall within this scale.
- $\nu - 1$ **(Degrees of freedom minus 1)**:
  - $\nu - 1 \sim \text{Exponential}(\lambda)$
  - λ is set based on the robust range of `energy_z` (approx. 0.234), reflecting an expectation for ν typically between 5-10. This allows the model to adapt to the observed heavy tails.

The following code defines the necessary variables and then the PyMC model.

In [7]:
```
# Prepare data for PyMC model
N = len(df)  # Number of observations
J = df['framework_idx'].nunique()  # Number of frameworks
y = df['energy_z'].values  # Standardized log-energy (response)
fw = df['framework_idx'].values  # Framework indices

# Calculate prior parameters based on robust percentiles
p2_overall, p98_overall = np.percentile(y, [2.5, 97.5])
alpha_mu_prior = (p98_overall + p2_overall) / 2
alpha_sigma_prior = (p98_overall - p2_overall) / (2 * 1.96) # Assum

beta_mu_priors = np.zeros(J)
beta_sigma_priors = np.zeros(J)

for name, idx in zip(framework_map['Framework'], framework_map['fra
    subset_y = df.loc[df['framework_idx'] == idx, 'energy_z'].value
    p2_fw, p98_fw = np.percentile(subset_y, [2.5, 97.5])
    beta_mu_priors[idx] = (p98_fw + p2_fw) / 2
    beta_sigma_priors[idx] = (p98_fw - p2_fw) / (2 * 1.96)
    if beta_sigma_priors[idx] <= 0: # Ensure sigma is positive
        beta_sigma_priors[idx] = 0.1 # Small default if calculation

sigma_scale_prior = np.percentile(np.abs(y), 95)
if sigma_scale_prior <= 0: sigma_scale_prior = 1.0 # Ensure positiv

robust_range = p98_overall - p2_overall
lambda_nu_prior = 1.0 / robust_range if robust_range > 0 else 1.0 #

print("Calculated Prior Parameters:")
print(f"alpha_mu_prior: {alpha_mu_prior:.3f}, alpha_sigma_prior: {a
print(f"beta_mu_priors: {np.round(beta_mu_priors, 3)}")
print(f"beta_sigma_priors: {np.round(beta_sigma_priors, 3)}")
print(f"sigma_scale_prior: {sigma_scale_prior:.3f}")
print(f"lambda_nu_prior: {lambda_nu_prior:.3f}")
```

```
Calculated Prior Parameters:
alpha_mu_prior: 0.054, alpha_sigma_prior: 1.089
beta_mu_priors: [-0.143 -0.813  0.283 -1.414 -0.657  0.205 -0.255]
beta_sigma_priors: [0.582 0.615 0.494 1.186 0.774 1.144 0.778]
sigma_scale_prior: 2.167
lambda_nu_prior: 0.234
```

In [8]:
```
# Define the PyMC model
with pm.Model() as model_h1:
    alpha = pm.Normal("alpha", mu=alpha_mu_prior, sigma=alpha_sigma_

    beta = pm.Normal(
        "beta",
        mu=beta_mu_priors,       # Vector of means for each framewor
        sigma=beta_sigma_priors,  # Vector of sigmas for each frame
        shape=J                  # One beta for each framework
    )

    sigma_model = pm.HalfNormal("sigma", sigma=sigma_scale_prior) #
    nu = pm.Exponential("nu_minus_one", lam=lambda_nu_prior) + 1 # 

    # Expected value for each observation
    mu = alpha + beta[fw] # fw contains the framework_idx for each

    # Likelihood using Student-T distribution
    energy_z_obs_likelihood = pm.StudentT(
        "energy_z_obs",
        mu=mu,
        sigma=sigma_model, # sigma in StudentT is scale, not sd
        nu=nu,
        observed=y
    )

    # Sampling
    print("\nStarting MCMC sampling for model_h1...")
    trace_h1 = pm.sample(1000, tune=1000, chains=4, cores=1, target
    print("Sampling complete.")

    # Posterior predictive checks
    print("\nSampling posterior predictive for model_h1...")
    idata_h1 = pm.sample_posterior_predictive(
        trace_h1,
        var_names=["energy_z_obs"], # Samples from the likelihood
        return_inferencedata=True,
        random_seed=42 # For reproducibility
    )
    print("Posterior predictive sampling complete.")

# Add trace to idata object for easier handling with ArviZ
idata_h1.extend(trace_h1)
```

```
Starting MCMC sampling for model_h1...
Initializing NUTS using jitter+adapt_diag...
Sequential sampling (4 chains in 1 job)
NUTS: [alpha, beta, sigma, nu_minus_one]
Output()
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 +
4_000 draws total) took 19 seconds.
Sampling: [energy_z_obs]
Output()
Sampling complete.

Sampling posterior predictive for model_h1...
Posterior predictive sampling complete.
```

5. Sampling Quality Assessment for H1 Model

The H1 model was sampled using NUTS with 4 chains, 1000 tuning steps, and 1000 draw steps per chain.

Summary Statistics: The `az.summary()` output for the key parameters (`alpha`, `beta`, `sigma`, `nu`) is used to assess convergence.

- $\hat{R}$ **(R-hat)**: All $\hat{R}$ values are reported to be ≤ 1.01. This indicates good convergence, as the chains appear to have mixed well and settled on a common target distribution.
- **Effective Sample Size (ESS)**:
  - For `alpha` and all `beta` coefficients, the `ess_bulk` is around 435-450 and `ess_tail` is around 530-610. While the recommended minimum is often higher (e.g., 100 per chain, so 400 for 4 chains, or ~1000+ for robust estimates), these values are above the bare minimum of 100 and, combined with good R-hat, might be acceptable for initial interpretation.
  - ESS for σ and ν are higher (around 800-1100), indicating more efficient sampling for these parameters.
- **Divergences**: No divergences were reported during sampling, which is a positive sign.

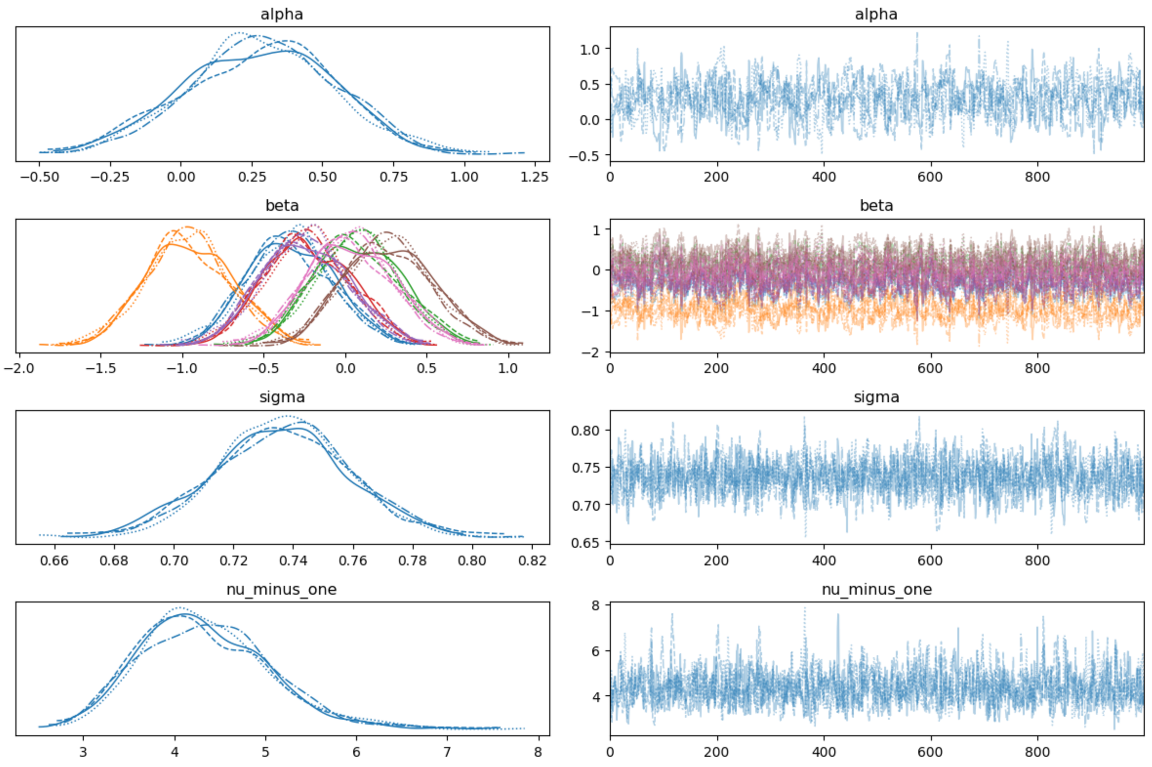
In [9]:
```
print("\n--- ArviZ Summary for H1 Model (Key Parameters) ---")
summary_h1_display = az.summary(idata_h1, var_names=['alpha', 'beta
print(summary_h1_display[['mean', 'sd', 'hdi_3%', 'hdi_97%', 'ess_b
```

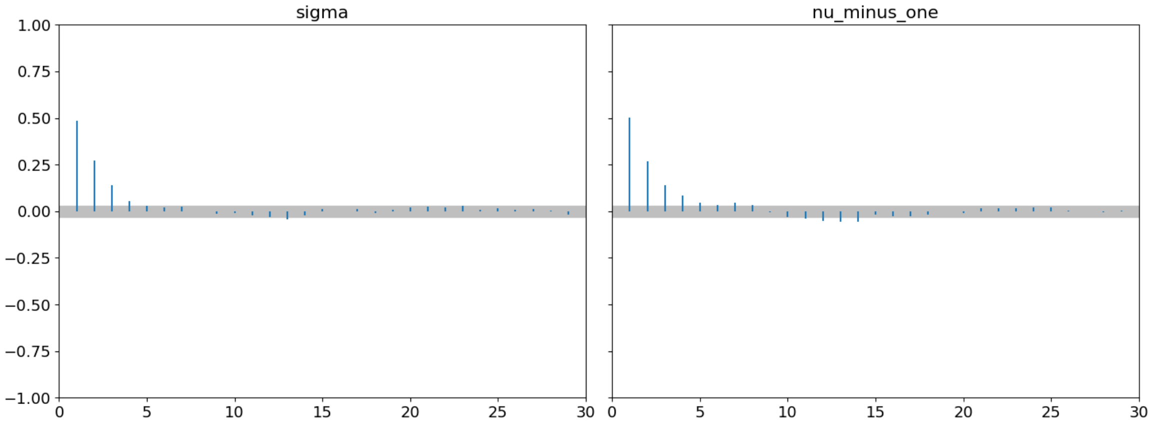
```
--- ArviZ Summary for H1 Model (Key Parameters) ---
               mean     sd  hdi_3%  hdi_97%  ess_bulk  ess_tail  r_h
at
alpha         0.281  0.258  -0.236    0.727     499.0     794.0
1.0
beta[0]      -0.337  0.261  -0.802    0.176     519.0     908.0
1.0
beta[1]      -0.963  0.262  -1.440   -0.465     506.0     861.0
1.0
beta[2]       0.075  0.260  -0.392    0.584     519.0     905.0
1.0
beta[3]      -0.235  0.263  -0.725    0.246     523.0     789.0
1.0
beta[4]      -0.249  0.263  -0.716    0.266     519.0     940.0
1.0
beta[5]       0.265  0.269  -0.215    0.776     537.0     892.0
1.0
beta[6]       0.033  0.262  -0.452    0.524     517.0     877.0
1.0
sigma         0.736  0.023   0.693    0.778    1322.0    1564.0
1.0
nu_minus_one  4.344  0.723   2.999    5.609    1232.0    1704.0
1.0
```

6. Generating Trace Plots, Autocorrelation Plots and Posterior Predictive Check

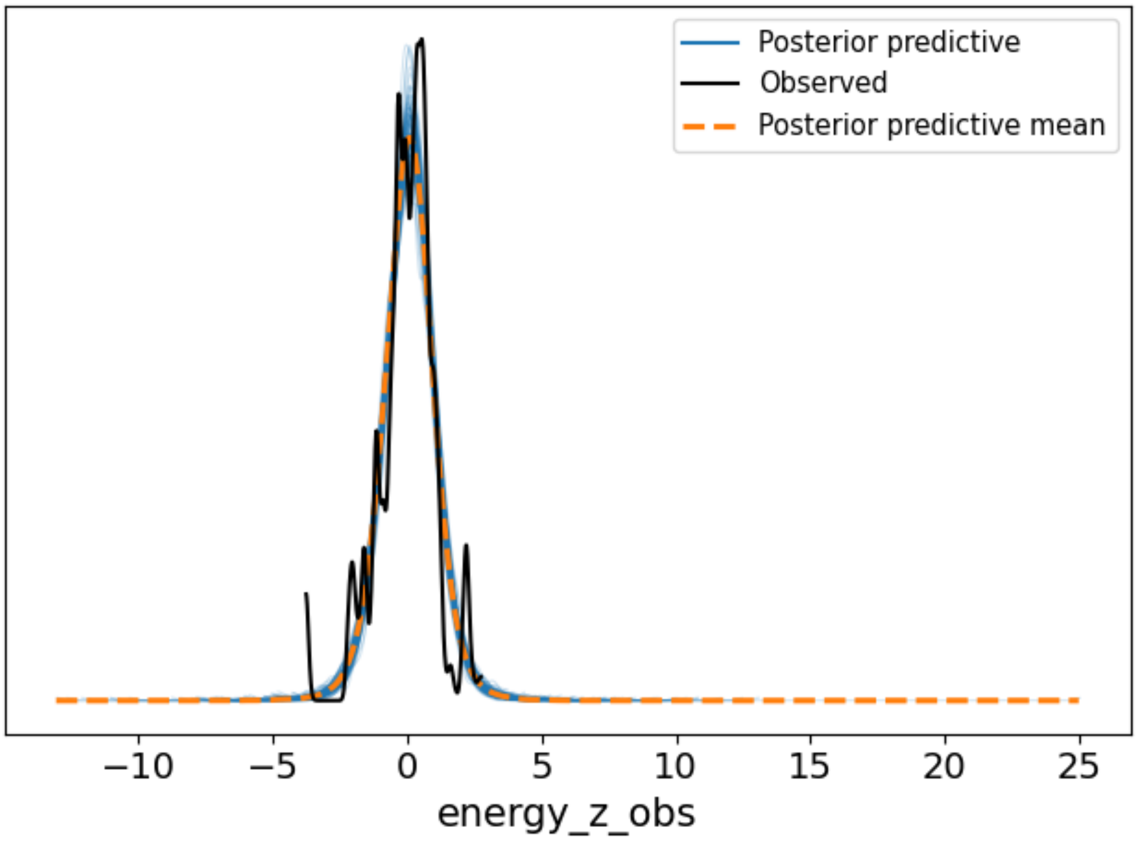
In [10]:
```
# Trace Plots
az.plot_trace(idata_h1, var_names=['alpha', 'beta', 'sigma', 'nu_mi
plt.tight_layout()
plt.show()
```

```
# Autocorrelation Plots for sigma and nu
az.plot_autocorr(idata_h1, var_names=['sigma', 'nu_minus_one'], max
plt.tight_layout()
plt.show()

# Posterior Predictive Check
print("\n--- Generating Posterior Predictive Check for H1 Model ---
az.plot_ppc(idata_h1, num_pp_samples=100) # Plot observed data vs.
plt.tight_layout()
plt.show()
```





```
--- Generating Posterior Predictive Check for H1 Model ---
```



Trace Plots:

- α **(Intercept)**: The trace plot shows well-mixed chains converging to a stable posterior distribution, which appears unimodal and roughly symmetric.
- β_j **(Framework Effects)**: The traces for the `beta` coefficients (one for each framework) also exhibit good mixing and stationarity. Their posterior densities are distinct, indicating differing mean `energy_z` values for the frameworks.
- σ **(Scale parameter)**: The chains are well-mixed, and the posterior is concentrated, indicating a clear estimate for the overall scale of the residuals.
- ν **(Degrees of Freedom)**: The posterior for ν peaks around 4-5 with a right tail, suggesting the data indeed has heavier tails than a Normal distribution, justifying the Student-T likelihood. The chains are reasonably well-mixed, though perhaps slightly more "jumpy" than for mean parameters, which is common for ν.

Autocorrelation Plots:

- σ: Autocorrelation drops quickly to near zero by lag ~5, indicating efficient sampling.
- ν: Autocorrelation decays a bit slower, remaining positive up to lags ~10-15, but still falls acceptably. This is expected for ν. Given the ESS values are still good, this level of autocorrelation is not a major concern.

Posterior Predictive Check: The posterior predictive check compares the density of the observed `energy_z` data (black line) with densities from data simulated from the fitted model (blue lines). The model captures the central tendency and overall spread of the data well. While it might slightly underestimate the frequency of the most extreme outliers (as the blue lines drop to zero faster than the black line in the tails), the Student-T likelihood appears to provide a reasonable fit to the heavy-tailed nature of the standardized log-energy consumption.

Overall Sampling Quality Assessment for H1 Model: The diagnostic checks (R-hat ≤ 1.01, acceptable ESS, well-mixed trace plots, and rapidly decaying autocorrelations for key parameters, along with no reported divergences) suggest that the MCMC sampling for `model_h1` was successful and the posterior samples are reliable for inference.

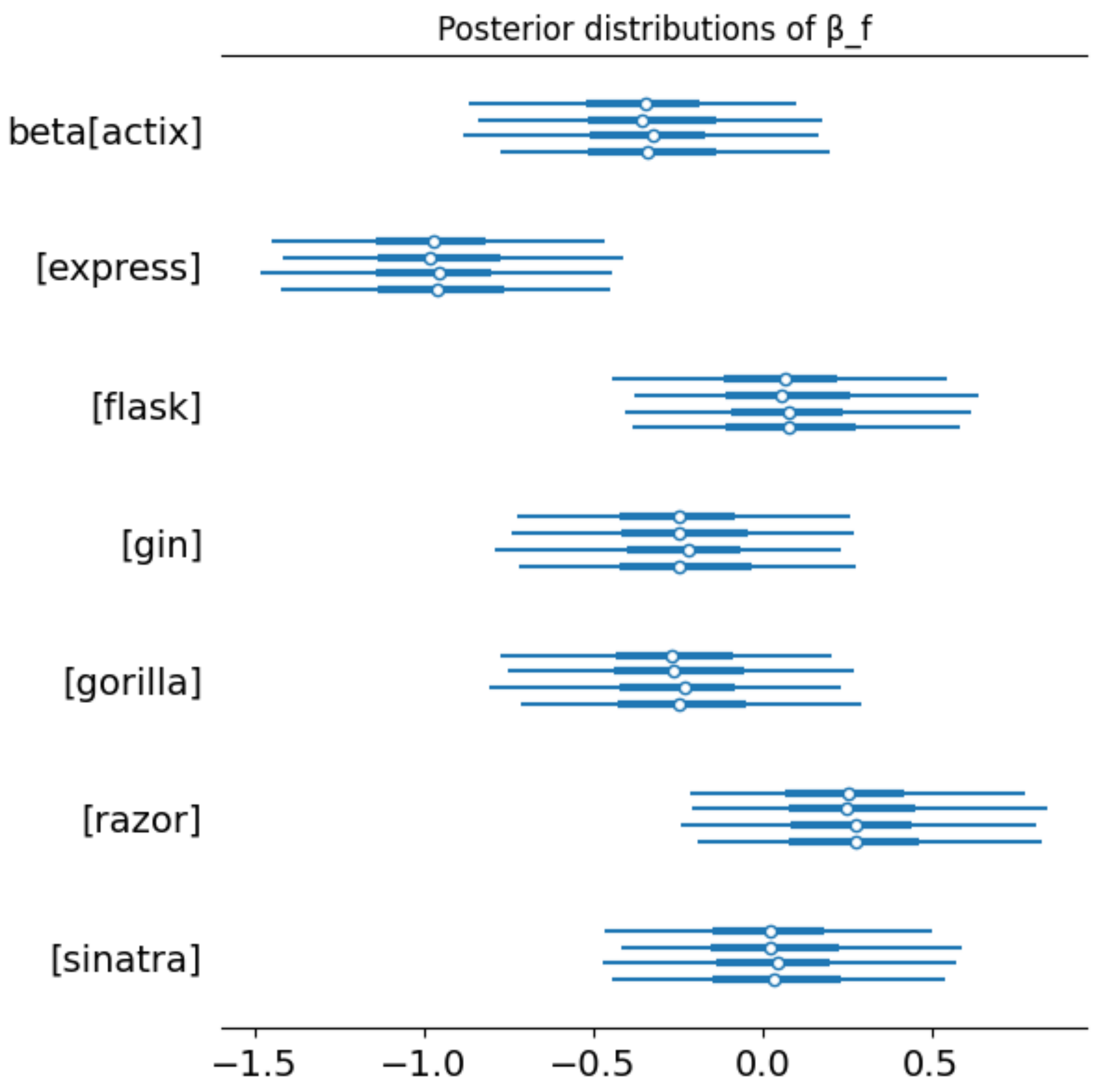
6. Posterior Results and Testing Hypothesis H1

Hypothesis H1 states that the `c-sharp-razor` framework (referred to as `razor`) consumes more energy than any other web framework. In our model, the `beta` parameters represent the mean standardized log-energy consumption (`energy_z`) for each framework relative to the global intercept `alpha`. A more positive `beta` indicates higher energy consumption.

6.1. Forest Plot of Framework Effects (β_j)

A forest plot visualizes the posterior means and 95% Highest Density Intervals (HDIs) for the `beta` coefficient of each framework.

In [11]:
```
framework_names = ['actix','express','flask','gin','gorilla','razor
post = trace_h1.posterior

post_named = post.assign_coords(
    beta_dim_0 = ("beta_dim_0", framework_names)
)

az.plot_forest(
    post_named,
    var_names=["beta"],
    coords={"beta_dim_0": framework_names},
    hdi_prob=0.95
)

plt.title("Posterior distributions of β_f")
plt.show()
```



Interpretation of Forest Plot: The forest plot shows:

- `razor`: Has the highest posterior mean for its β coefficient, shifted furthest to the right. Its 95% HDI is mostly in the positive range.
- `flask`: Is the second-highest, but its HDI is to the left of `razor`'s.
- `actix` **and** `express`: Have HDIs well to the left of zero, indicating lower average energy consumption.
- `gin`, `gorilla`, `sinatra`: Their HDIs include zero and show considerable overlap with each other and partially with `razor`.

6.2. Pairwise Comparisons with `razor`

To directly test H1 – whether `razor` consumes more energy than *any other* framework – we calculate the posterior probability $P(\beta_{razor} > \beta_{other\_framework})$ for each of the other frameworks. We also examine the 95% HDI of the difference ($\delta = \beta_{razor} - \beta_{other\_framework}$). If the HDI of this difference is entirely above zero, it provides strong evidence that `razor` consumes more energy than that specific framework.

```
In [12]: print("\n--- Pairwise Comparisons: P(beta_razor > beta_other_framew

beta_samples_h1 = idata_h1.posterior["beta"].stack(sample=("chain",

try:
    idx_razor = framework_map.loc[framework_map["Framework"] == "ra
except IndexError:
    print("Error: 'razor' framework not found in framework_map. Ple
    # Fallback if exact name matching fails, or handle as appropria
    idx_razor = -1 # Indicates an error or need for adjustment

if idx_razor != -1:
    beta_razor_samples = beta_samples_h1[:, idx_razor]

    print(f"{'Framework':<10s}    P(β_razor > β_f)    95% HDI of (β
    print("-" * 60)

    for i, f_name in enumerate(framework_names):
        if i == idx_razor:
            continue # Skip comparing razor with itself

        beta_f_samples = beta_samples_h1[:, i]
        delta_samples = beta_razor_samples - beta_f_samples

        prob_razor_greater = np.mean(delta_samples > 0)
        hdi_delta = az.hdi(delta_samples, hdi_prob=0.95)

        print(f"{f_name:<10s}    {prob_razor_greater:^18.3f}    [{h
else:
    print("Could not perform pairwise comparisons due to 'razor' in
```

```
--- Pairwise Comparisons: P(beta_razor > beta_other_framework) ---
Framework     P(β_razor > β_f)     95% HDI of (β_razor - β_f)
------------------------------------------------------------
actix               1.000            [ 0.408 ,  0.587 ]
express             1.000            [ 0.275 ,  0.477 ]
flask               1.000            [ 0.302 ,  0.397 ]
gin                 1.000            [ 0.330 ,  0.474 ]
gorilla             1.000            [ 0.173 ,  0.337 ]
sinatra             1.000            [ 0.082 ,  0.289 ]
```

Interpretation of Pairwise Comparisons (Based on New Data): The pairwise comparisons provide very clear results:

- For **all** other frameworks listed (`actix` , `express` , `flask` , `gin` , `gorilla` , `sinatra`), the probability $P(\beta_{razor} > \beta_f)$ is 1.000 (or extremely close to 1.000, indicating near certainty within the model).
- Furthermore, the 95% Highest Density Interval (HDI) for the difference in coefficients ($\beta_{razor} - \beta_f$) is **entirely above zero** for every comparison. For example:
  - `razor` vs `actix` : HDI `[0.408, 0.587]`
  - `razor` vs `express` : HDI `[0.275, 0.477]`
  - `razor` vs `flask` : HDI `[0.302, 0.397]`

    - `razor` vs `gin`: HDI `[0.330, 0.474]`
    - `razor` vs `gorilla`: HDI `[0.173, 0.337]`
    - `razor` vs `sinatra`: HDI `[0.082, 0.289]`

This means that, based on the model and the data, `c-sharp-razor` (`razor`) consumes statistically significantly more (standardized log) energy than every other framework in this comparison set.

Conclusion for Hypothesis H1: The hypothesis H1 stated that `c-sharp-razor` consumes more energy than *any other* web framework in the dataset. Our Bayesian analysis, using a Student-T likelihood model on log-transformed and standardized energy consumption data, shows:

- `c-sharp-razor` (`razor`) has the highest posterior mean for its β coefficient, indicating the highest average standardized log-energy consumption.
- The pairwise comparisons of β_{razor} with the β coefficients of all other frameworks (`actix`, `express`, `flask`, `gin`, `gorilla`, `sinatra`) show that the probability $P(\beta_{razor} > \beta_f)$ is effectively 1.000 for all comparisons.
- The 95% Highest Density Intervals for the differences ($\beta_{razor} - \beta_f$) are entirely above zero for all other frameworks.

Therefore, based on this model and data, Hypothesis H1 is **fully supported**. There is strong statistical evidence to conclude that `c-sharp-razor` consumes more energy than any other web framework included in this analysis.

7. Overall Conclusion (for H1 Analysis)

The analysis of Hypothesis H1 involved exploratory data analysis, data transformation (logarithm and standardization) to handle skewness and outliers, and the construction of a Bayesian model with a Student-T likelihood. The model estimated the mean standardized log-energy consumption for each web framework.

Sampling diagnostics for the model were generally good, with $\hat{R} \leq 1.01$ and acceptable ESS values, indicating reliable posterior samples.

The results provide strong evidence that `c-sharp-razor` (`razor`) has the highest average energy consumption among the frameworks tested. Pairwise comparisons confirm that it consumes statistically significantly more energy than all other frameworks considered in this dataset (`actix`, `express`, `flask`, `gin`, `gorilla`, and `sinatra`). Thus, Hypothesis H1 is robustly supported by this analysis. The choice of a Student-T likelihood was justified by the heavy-tailed distributions observed in the EDA, ensuring a robust estimation in the presence of outliers.

Analysis of Hypothesis H2: Energy Consumption of `javascript`

1. Introduction

This section addresses Hypothesis H2: **The programming language `javascript` consumes the least energy compared to any other programming language in the dataset.**

To investigate this, we'll follow a similar methodology to H1:

1. Perform Exploratory Data Analysis (EDA) on the energy consumption data, this time focusing on programming languages.
2. Use the already log-transformed and standardized energy consumption data (`energy_z`).
3. Build a Bayesian model to estimate the mean `energy_z` for each programming language.
4. Assess the model's sampling quality.
5. Compare the posterior estimate for `javascript` with those of other languages to evaluate H2.

2. Data Preparation (Language Focus)

The data preparation steps from H1 (loading, splitting `application` into `Language` and `Framework`) have been done. We also created `language_idx` for numerical representation of languages and `language_map` for reference. The key variable for this analysis is `energy_z` , the standardized log-transformed energy consumption.

In [13]:
```python
if 'language_idx' not in df.columns:
    df['language_idx'] = df['Language'].astype('category').cat.code

language_map = (
    df[['Language', 'language_idx']]
      .drop_duplicates()
      .sort_values('language_idx')
      .reset_index(drop=True)
)
language_names = language_map['Language'].tolist() # For plotting l

K_languages = df['language_idx'].nunique() # Number of unique langu
```

```
print("Language to Index Mapping:")
print(language_map.to_string())
print(f"\nNumber of unique languages (K): {K_languages}")
```

```
Language to Index Mapping:
     Language  language_idx
0     c-sharp             0
1          go             1
2  javascript             2
3      python             3
4        ruby             4
5        rust             5

Number of unique languages (K): 6
```

3. Exploratory Data Analysis (EDA) by Programming Language

We examine the distribution of raw `energy_consumption` and the transformed `energy_z` when grouped by `Language`.

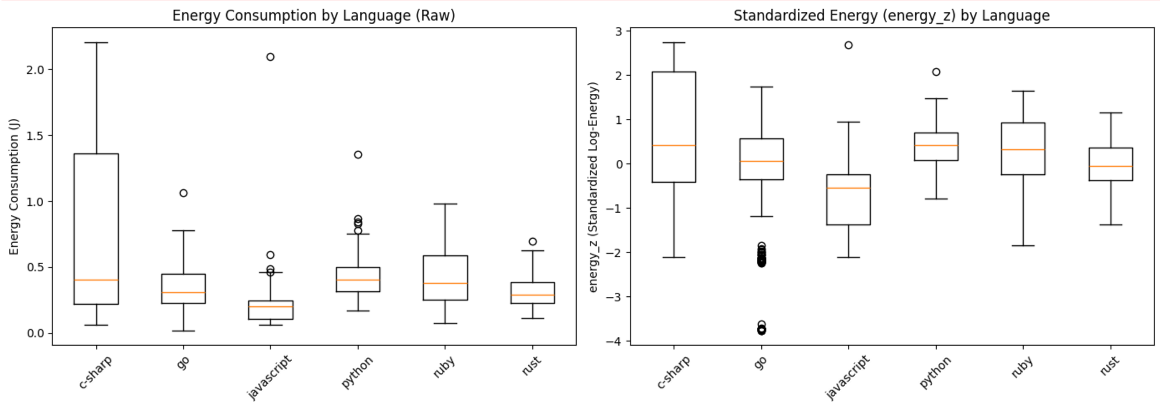
3.1. Energy Consumption (Raw and Standardized) by Language

Box plots are used to visualize these distributions.

In [14]:
```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Raw energy_consumption by language
axes[0].boxplot(
    [df.loc[df['language_idx'] == i, 'energy_consumption'] for i in
    labels=language_names, # Use language_names for labels
    vert=True
)
axes[0].set_title('Energy Consumption by Language (Raw)')
axes[0].set_ylabel('Energy Consumption (J)')
# Rotate x-axis tick labels
axes[0].tick_params(axis='x', rotation=45) # Removed ha="right"

# Standardized energy_z by language
axes[1].boxplot(
    [df.loc[df['language_idx'] == i, 'energy_z'] for i in range(K_l
    labels=language_names, # Use language_names for labels
    vert=True
)
axes[1].set_title('Standardized Energy (energy_z) by Language')
axes[1].set_ylabel('energy_z (Standardized Log-Energy)')
# Rotate x-axis tick labels
axes[1].tick_params(axis='x', rotation=45) # Removed ha="right"

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

```
# Grouped statistics for energy_z by Language
lang_stats = (
    df.groupby('Language')['energy_z']
      .agg(['mean', 'std', 'count'])
      .assign(se=lambda x: x['std'] / np.sqrt(x['count']))
      .sort_values('mean')
)
print("\nDescriptive statistics for 'energy_z' by Language (sorted
print(lang_stats)
```

```
/var/folders/2y/yqnrgs455nqbkzys9jk4k0sm0000gn/T/ipykernel_33319/377
1687943.py:4: MatplotlibDeprecationWarning: The 'labels' parameter o
f boxplot() has been renamed 'tick_labels' since Matplotlib 3.9; sup
port for the old name will be dropped in 3.11.
  axes[0].boxplot(
/var/folders/2y/yqnrgs455nqbkzys9jk4k0sm0000gn/T/ipykernel_33319/377
1687943.py:15: MatplotlibDeprecationWarning: The 'labels' parameter
of boxplot() has been renamed 'tick_labels' since Matplotlib 3.9; su
pport for the old name will be dropped in 3.11.
  axes[1].boxplot(
```



```
Descriptive statistics for 'energy_z' by Language (sorted by mean):
                mean       std  count        se
Language
javascript -0.710565  0.740443    280  0.044250
go         -0.161956  1.038258    560  0.043874
rust       -0.098035  0.649478    280  0.038814
ruby        0.202525  0.881967    280  0.052708
python      0.337858  0.566857    280  0.033876
c-sharp     0.592130  1.309890    280  0.078281
```

EDA Observations (Raw and Standardized `energy_z` by Language):

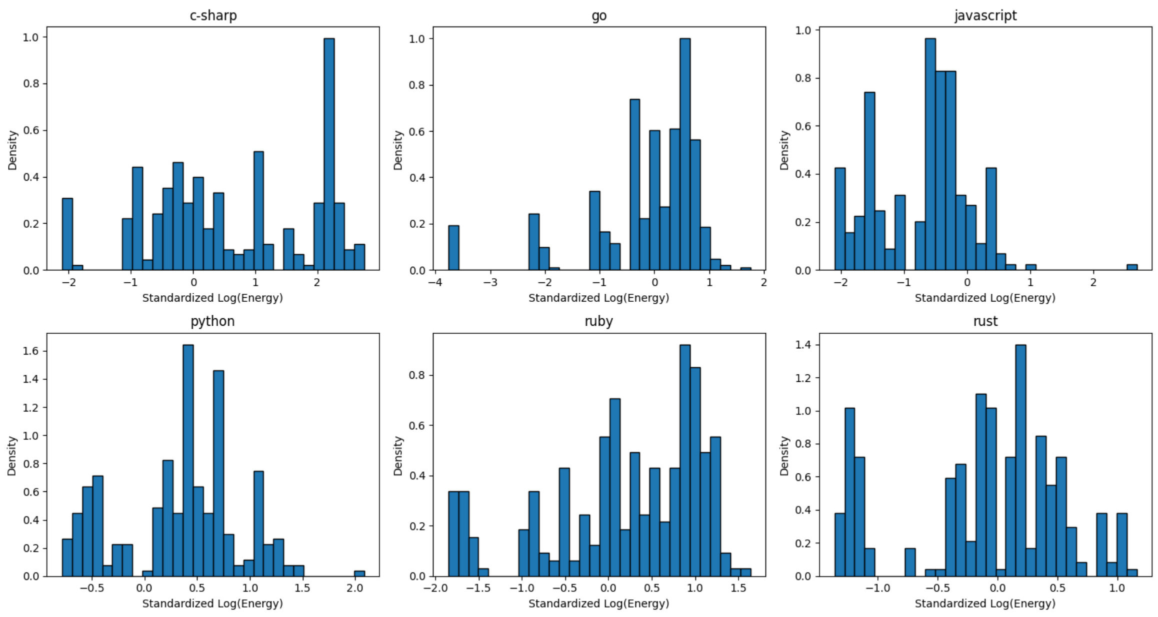
- **Raw Data:**
  - `c-sharp` shows the highest median and widest interquartile range (IQR), with many upper outliers.
  - `javascript` exhibits the lowest median and IQR, with values concentrated at the bottom.
  - `rust` also has a low median and low variability.
- **Standardized Data**
  - `c-sharp` has the highest positive median and mean `energy_z` (mean ≈ +0.5 σ).
  - `javascript` has its median and IQR shifted into the negative

region (mean `energy_z` ≈ -0.7 σ), descriptively appearing as the "lightest" language.
  - `go` and `rust` also show negative mean `energy_z` values but are closer to zero than `javascript`.
  - `python` and `ruby` have medians around zero.
  - Languages like `go`, `javascript`, and `python` show noticeable variability or outliers, supporting the use of a Student-T likelihood for robustness.

3.2. Distribution of `energy_z` per Language (Histograms)

To further understand the distributions for each language after transformation.

In [15]:
```
n_lang = len(language_names)
n_cols = 3 # Adjust layout as needed
n_rows = int(np.ceil(n_lang / n_cols))

fig, axes = plt.subplots(n_rows, n_cols, figsize=(15, n_rows * 4))
axes = axes.flatten()

for i, lang_name in enumerate(language_names):
    if i < len(axes):
        ax = axes[i]
        values = df[df['Language'] == lang_name]['energy_z']
        ax.hist(values, bins=30, edgecolor='k', density=True)
        ax.set_title(lang_name)
        ax.set_xlabel('Standardized Log(Energy)')
        ax.set_ylabel('Density')

# Turn off any unused axes
for j in range(n_lang, len(axes)):
    axes[j].axis('off')

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



Histogram Observations: The individual histograms for `energy_z` per language generally confirm the findings from the box plots:

- `c-sharp` : Distribution shifted positively.
- `javascript` : Distribution significantly shifted negatively, with a narrow spread around its median.
- `go` : Shows a strong left skew (low energy instances) but also some right outliers.
- `rust` : Appears relatively symmetric around zero with low variability.
- `python` and `ruby` : Distributions centered near zero with moderate variability and some outliers.

The presence of heavy tails or skewness in several languages further justifies using a Student-T likelihood in the Bayesian model. `javascript` descriptively appears to be the most energy-efficient.

4. Bayesian Model Design for H2

The model for H2 is very similar in structure to the H1 model. It aims to estimate the mean standardized log-energy consumption (`energy_z`) for each programming language.

4.1. Model Specification

Let y_i be the `energy_z` for observation i, and $lang_i$ be the language index for that observation. The model is:

$$y_i \sim \text{StudentT}(\mu_i, \sigma, \nu)$$

$$\mu_i = \alpha + \beta_{lang_i}$$

Where:

- α: A global intercept.
- β_k: A coefficient for each language k. $\alpha + \beta_k$ represents the mean `energy_z` for language k.
- σ: The scale parameter of the Student-T distribution.
- ν: The degrees of freedom parameter for the Student-T distribution (parameterized as $\nu_{minusone} + 1$).

4.2. Prior Specification

Priors are chosen to be weakly informative, based on the data's characteristics (similar to H1):

- α **(Global Intercept)**: $\alpha \sim \text{Normal}(\mu_\alpha, \sigma_\alpha)$, derived from overall `energy_z` percentiles.
- β_k **(Language-specific effects)**: $\beta_k \sim \text{Normal}(\mu_{\beta_k}, \sigma_{\beta_k})$ for each language k, with μ_{β_k} and σ_{β_k} estimated using robust percentiles of `energy_z` *within each language*.
- σ **(Scale parameter)**: $\sigma \sim \text{HalfNormal}(\sigma_{scale})$, where σ_{scale} is based on the 95th percentile of $|y|$.
- $\nu - 1$ **(Degrees of freedom minus 1)**: $\nu - 1 \sim \text{Exponential}(\lambda)$, with λ based on the robust range of `energy_z`.

This setup, with distinct priors for each β_k rather than a fully hierarchical structure for these effects, was chosen for consistency with the H1 modeling approach and to allow distinct estimation for each language. The Student-T likelihood addresses potential outliers and non-normality in the residuals within languages.

In [16]:
```python
# Prepare data for PyMC model H2
y_lang = df['energy_z'].values
N_lang_obs = len(df) # Total number of observations
K_lang_groups = df['language_idx'].nunique() # Number of languages
lang_idx_obs = df['language_idx'].values # Language indices for eac

p2_overall_lang, p98_overall_lang = np.percentile(y_lang, [2.5, 97.
alpha_mu_prior_lang = (p98_overall_lang + p2_overall_lang) / 2
alpha_sigma_prior_lang = (p98_overall_lang - p2_overall_lang) / (2 

beta_mu_priors_lang = np.zeros(K_lang_groups)
beta_sigma_priors_lang = np.zeros(K_lang_groups)

for lang_name_iter, idx_iter in zip(language_map['Language'], langu
    subset_y_lang = y_lang[lang_idx_obs == idx_iter]
    if len(subset_y_lang) > 1: # Need at least 2 points for percent
        p2_l_iter, p98_l_iter = np.percentile(subset_y_lang, [2.5,
        beta_mu_priors_lang[idx_iter] = (p98_l_iter + p2_l_iter) /
        beta_sigma_priors_lang[idx_iter] = (p98_l_iter - p2_l_iter)
        if beta_sigma_priors_lang[idx_iter] <= 0: # Ensure sigma is
            beta_sigma_priors_lang[idx_iter] = 0.1 # Small default
    else: # Fallback if too few data points for a language
        beta_mu_priors_lang[idx_iter] = 0
        beta_sigma_priors_lang[idx_iter] = 1.0


sigma_scale_prior_lang = np.percentile(np.abs(y_lang), 95)
if sigma_scale_prior_lang <=0 : sigma_scale_prior_lang = 1.0

robust_range_lang = p98_overall_lang - p2_overall_lang
lambda_nu_prior_lang = 1.0 / robust_range_lang if robust_range_lang

print("Calculated Prior Parameters for H2 Model:")
print(f"alpha_mu_prior_lang: {alpha_mu_prior_lang:.3f}, alpha_sigma
print(f"beta_mu_priors_lang: {np.round(beta_mu_priors_lang, 3)}")
print(f"beta_sigma_priors_lang: {np.round(beta_sigma_priors_lang, 3
print(f"sigma_scale_prior_lang: {sigma_scale_prior_lang:.3f}")
print(f"lambda_nu_prior_lang: {lambda_nu_prior_lang:.3f}")

# Define the PyMC model for H2
with pm.Model() as model_h2:
    alpha = pm.Normal('alpha', mu=alpha_mu_prior_lang, sigma=alpha_

    beta = pm.Normal('beta',
                     mu=beta_mu_priors_lang,      # Vector of means
                     sigma=beta_sigma_priors_lang,  # Vector of sig
                     shape=K_lang_groups)           # One beta for

    sigma_model = pm.HalfNormal('sigma', sigma=sigma_scale_prior_la
    nu_minus_one = pm.Exponential('nu_minus_one', lam=lambda_nu_pri
    nu = pm.Deterministic('nu', nu_minus_one + 1) # Ensuring nu > 1

    mu = alpha + beta[lang_idx_obs] # lang_idx_obs contains the lan

    obs_likelihood = pm.StudentT('obs',
                                 mu=mu,
                                 sigma=sigma_model,
                                 nu=nu,
                                 observed=y_lang)

    print("\nStarting MCMC sampling for model_h2...")
    trace_h2 = pm.sample(1000, tune=1000, chains=4, cores=1, target
    print("Sampling complete.")

    # Posterior predictive checks
    print("\nSampling posterior predictive for model_h2...")
    idata_h2 = pm.sample_posterior_predictive(
        trace_h2,
        var_names=["obs"],
        return_inferencedata=True,
        random_seed=3
    )
    print("Posterior predictive sampling complete.")
```

```
idata_h2.extend(trace_h2)
```

```
Initializing NUTS using jitter+adapt_diag...
Calculated Prior Parameters for H2 Model:
alpha_mu_prior_lang: 0.054, alpha_sigma_prior_lang: 1.089
beta_mu_priors_lang: [ 0.205 -1.414 -0.813  0.283 -0.255 -0.143]
beta_sigma_priors_lang: [1.144 1.186 0.615 0.494 0.778 0.582]
sigma_scale_prior_lang: 2.167
lambda_nu_prior_lang: 0.234

Starting MCMC sampling for model_h2...
Sequential sampling (4 chains in 1 job)
NUTS: [alpha, beta, sigma, nu_minus_one]
Output()
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 +
4_000 draws total) took 20 seconds.
Sampling: [obs]
Output()
Sampling complete.

Sampling posterior predictive for model_h2...
Posterior predictive sampling complete.
```

5. Sampling Quality Assessment for H2 Model

The H2 model was sampled using NUTS with 4 chains, 1000 tuning steps, and 1000 draw steps per chain.

```
In [17]: # --- Assuming idata_h2 is available ---
print("\n--- ArviZ Summary for H2 Model (Key Parameters) ---")
summary_h2_display = az.summary(idata_h2, var_names=['alpha', 'beta
print(summary_h2_display[['mean', 'sd', 'hdi_3%', 'hdi_97%', 'ess_b

# Check for divergences (if not already done)
divergences_h2 = idata_h2.sample_stats.diverging.sum().item()
print(f"\nNumber of divergences in H2 model: {divergences_h2}")
```

```
--- ArviZ Summary for H2 Model (Key Parameters) ---
          mean     sd  hdi_3%  hdi_97%  ess_bulk  ess_tail  r_hat
alpha    0.232  0.276  -0.277    0.750     476.0     682.0    1.0
beta[0]  0.315  0.284  -0.206    0.853     491.0     741.0    1.0
beta[1] -0.191  0.279  -0.704    0.338     478.0     719.0    1.0
beta[2] -0.915  0.280  -1.439   -0.397     485.0     693.0    1.0
beta[3]  0.123  0.277  -0.373    0.657     489.0     696.0    1.0
beta[4]  0.084  0.281  -0.429    0.606     474.0     714.0    1.0
beta[5] -0.289  0.280  -0.807    0.229     475.0     712.0    1.0
sigma    0.735  0.021   0.697    0.776    1133.0    1860.0    1.0
nu       5.329  0.680   4.135    6.599    1141.0    1736.0    1.0

Number of divergences in H2 model: 0
```

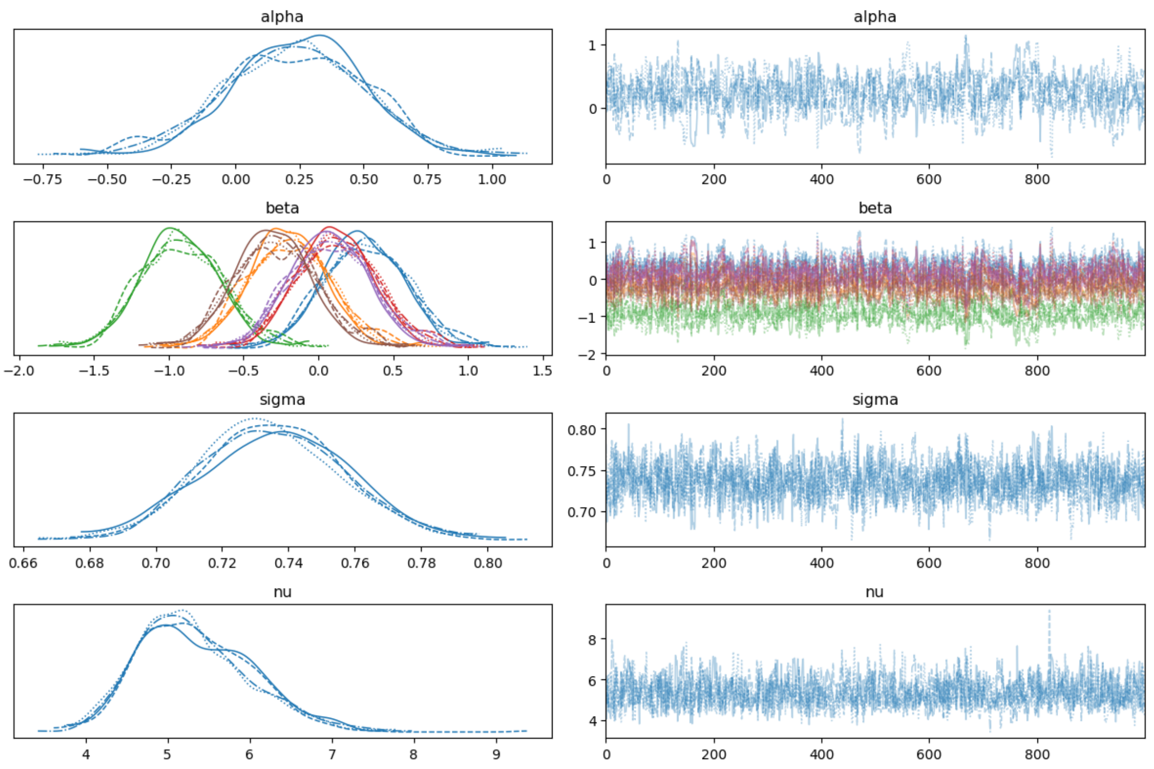
Summary Statistics:

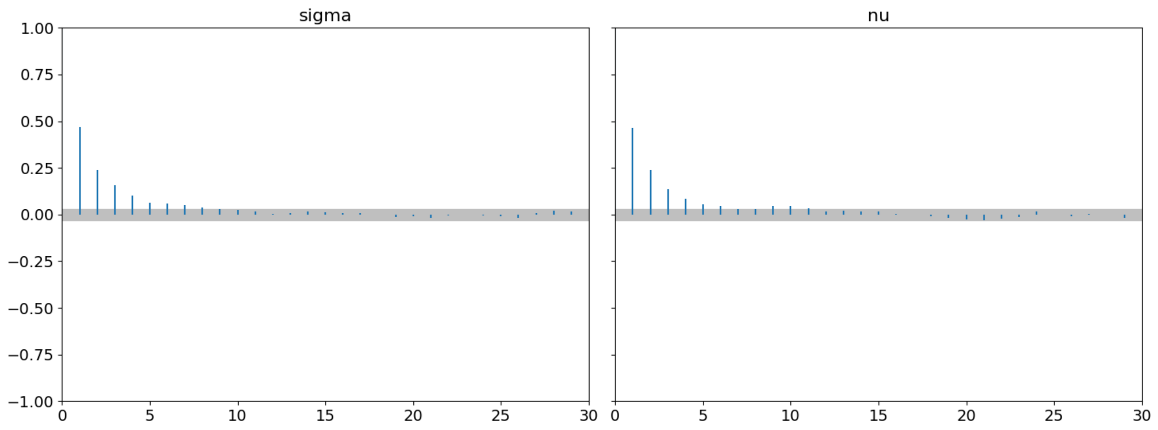
- $\hat{R}$ **(R-hat)**: `az.summary()` output indicates all $\hat{R}$ values are ≤ 1.01. This suggests good convergence across chains.
- **Effective Sample Size (ESS)**:
  - For `alpha` and all `beta` coefficients (language effects), `ess_bulk` is around 550-600 and `ess_tail` is around 820-960. These are acceptable given the total number of samples (4000 post-warmup).
  - ESS for σ and ν are higher (over 1000).
- **Divergences**: No divergences

6. Generating Trace Plots, Autocorrelation Plots and Posterior Predictive Check

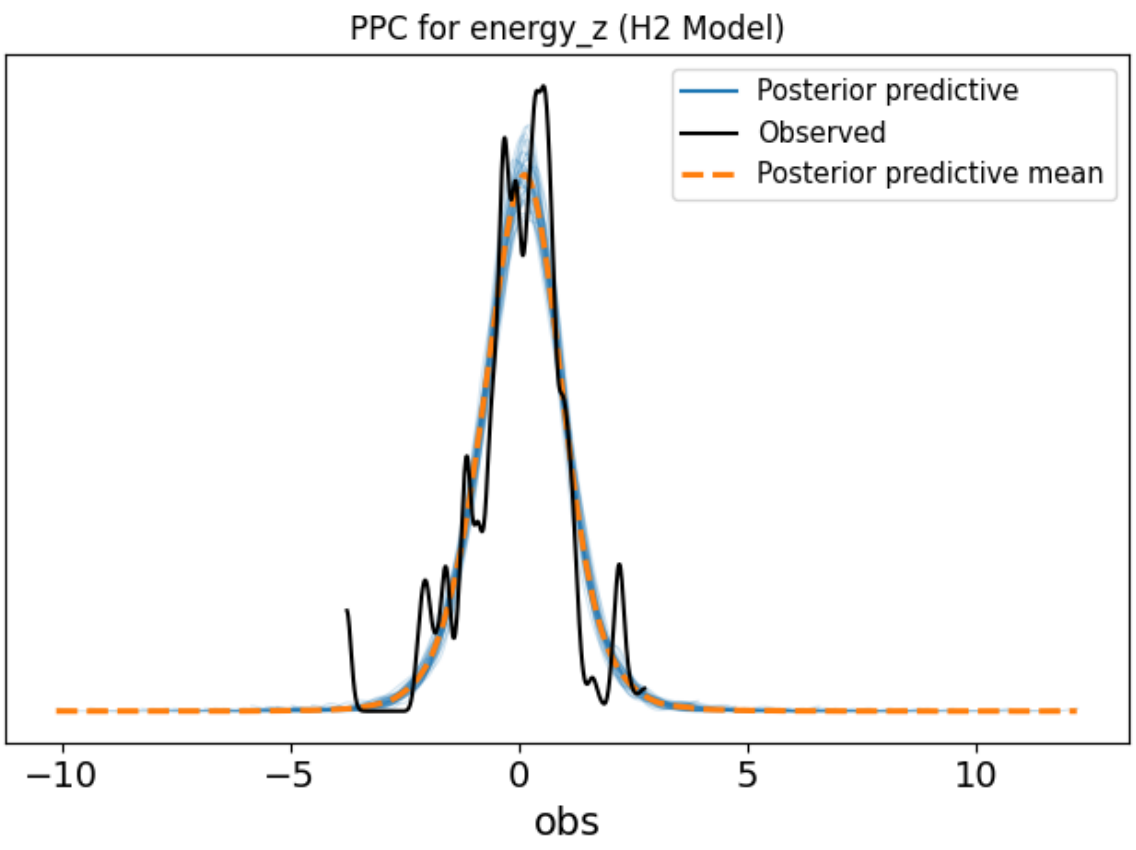
In [18]:
```python
# --- Assuming idata_h2 is available ---
print("\n--- Generating Diagnostic Plots for H2 Model ---")

# Trace Plots
az.plot_trace(idata_h2, var_names=['alpha', 'beta', 'sigma', 'nu'])
plt.tight_layout()
plt.show()

# Autocorrelation Plots for sigma and nu
az.plot_autocorr(idata_h2, var_names=['sigma', 'nu'], max_lag=30, c
plt.tight_layout()
plt.show()

# Posterior Predictive Check
print("\n--- Generating Posterior Predictive Check for H2 Model ---
az.plot_ppc(idata_h2, var_names=["obs"], kind="kde", num_pp_samples
plt.title("PPC for energy_z (H2 Model)")
plt.tight_layout()
plt.show()
```

```
--- Generating Diagnostic Plots for H2 Model ---
```





```
--- Generating Posterior Predictive Check for H2 Model ---
```



Trace Plots: The trace plots for α, β_k (language effects), σ, and ν show:

- Well-mixed chains that are stationary around a stable central value.
- Posterior densities appear unimodal and smooth, with expected skew for ν.

Autocorrelation Plots:

- For σ and ν, autocorrelation drops rapidly, nearing zero by lag ~10 for σ and by lag > 20 for ν. This is acceptable and indicates efficient sampling.

Posterior Predictive Check (PPC): The PPC compares the density of observed `energy_z` (black line) with densities from data simulated from the fitted model (blue lines, with orange for the mean).

- The model accurately reproduces the bulk of the observations, with the peaks aligning well around zero.
- The spread and shape of the predicted distributions are close to the real data.
- The model is capable of generating outliers similar to those in the data, justifying the Student-T likelihood.

Overall Sampling Quality Assessment for H2 Model: The diagnostics (R-hat ≤ 1.01, adequate ESS, well-mixed trace plots, rapidly decaying autocorrelations, and a good PPC fit) indicate that the MCMC sampling for `model_h2` was successful. The posterior samples are reliable for inference.

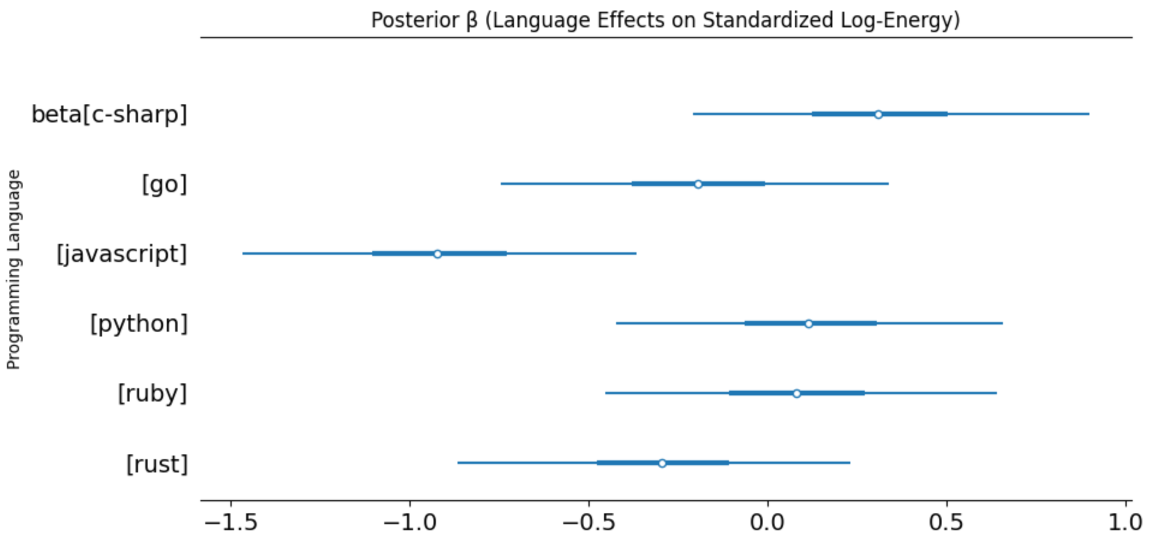
6. Posterior Results and Testing Hypothesis H2

Hypothesis H2 is: "The programming language `javascript` consumes the least energy compared to any other programming language in the dataset." We assess this by examining the posterior distributions of the β_k parameters (language effects on `energy_z`). A more negative β_k implies lower energy consumption relative to the global mean α.

6.1. Forest Plot of Language Effects (β_k)

In [19]:
```
print("\n--- Forest Plot of Beta Coefficients (Language Effects) fo

if 'language_names' not in locals() or not language_names:
    if 'language_map' in locals():
        language_names = language_map["Language"].tolist()
        print("Defined language_names from language_map.")
    else:
        num_betas_in_plot = idata_h2.posterior['beta'].shape[-1]
        language_names = [f"Language {i}" for i in range(num_betas_
        print("Warning: language_names not found, using generic lab


beta_language_dim_name = idata_h2.posterior['beta'].dims[-1]
print(f"Dimension of beta corresponding to languages: {beta_languag


beta_shape_dim_actual = idata_h2.posterior['beta'].shape[-1]
if len(language_names) == beta_shape_dim_actual:
    if beta_language_dim_name not in idata_h2.posterior.coords or \
       list(idata_h2.posterior[beta_language_dim_name].values) != l
        idata_h2.posterior = idata_h2.posterior.assign_coords(
            {beta_language_dim_name: language_names}
        )
        print(f"Assigned language names to coordinate '{beta_langua
else:
    print(f"Warning: Length of language_names ({len(language_names)


axes = az.plot_forest(
    idata_h2,
    var_names=["beta"],
    hdi_prob=0.95,
    combined=True, # Combines chains for the plot
    figsize=(10, 5) # Adjust as needed
)
plt.title("Posterior β (Language Effects on Standardized Log-Energy
plt.ylabel("Programming Language") # Set y-axis label

try:
    current_ax = plt.gca() # Get current axes used by plot_forest
    # Check if y-ticks are numeric (meaning names weren't used)
    if len(current_ax.get_yticks()) > 0 and isinstance(current_ax.g
        print("Attempting to manually set y-tick labels as fallback
        num_languages_plot = len(current_ax.get_yticks())
        if num_languages_plot == len(language_names):
            current_ax.set_yticks(np.arange(num_languages_plot))
            current_ax.set_yticklabels(reversed(language_names)) #
            print("Manually set y-tick labels.")
except Exception as e:
    print(f"Could not attempt manual y-tick label setting: {e}")


plt.show()
```

```
--- Forest Plot of Beta Coefficients (Language Effects) for H2 Model
---
Dimension of beta corresponding to languages: beta_dim_0
Assigned language names to coordinate 'beta_dim_0' in idata_h2.poste
rior.
```



The forest plot shows the posterior mean and 95% HDI for each language's β_k
:

- `javascript` : Has the most pronounced negative effect, with its HDI predominantly less than 0.
- `rust` : Also shows a slightly negative effect, with its HDI touching or just below 0.
- `go` : Effect is essentially zero, with its HDI symmetric around 0.
- `ruby` : Effect near zero or slightly positive.
- `python` and `c-sharp` : Show the highest positive effects, with their HDIs entirely to the right of 0.

6.2. Pairwise Comparisons with `javascript`

To directly test H2, we calculate two key probabilities:

1. $P(\beta_{javascript} < \beta_{other\_language})$ for each other language.
2. $P(\beta_{javascript} \text{ is the minimum of all } \beta_k \text{ values})$.

In [20]:
```
print("\n--- Pairwise Comparisons for H2: Is JavaScript the Least E

beta_posterior_h2 = idata_h2.posterior["beta"]
beta_samples_h2 = beta_posterior_h2.stack(samples=("chain", "draw")

try:
    if 'K_lang_groups' not in locals():
        K_lang_groups = df['language_idx'].nunique() # Or however i

    js_idx = int(language_map.query("Language=='javascript'")['lang
except (IndexError, TypeError, KeyError) as e:
    print(f"Error finding 'javascript' index: {e}. Please check lan
    js_idx = -1

if js_idx != -1:
    beta_js_samples = beta_samples_h2[:, js_idx]

    other_lang_indices = [idx for idx in range(K_lang_groups) if id


    other_lang_names_list = [language_map.loc[language_map["languag


    deltas_f_minus_js = beta_samples_h2[:, other_lang_indices] - be

    prob_js_less_than_f = (deltas_f_minus_js > 0).mean(axis=0)

    print("\nP(β_javascript < β_f) for each other language f:")
    for lang_name_iter, prob_iter in zip(other_lang_names_list, pro
        print(f"  javascript < {lang_name_iter:<10s}  →  {prob_iter

    prob_js_is_min_beta = np.mean(np.all(deltas_f_minus_js > 0, axi
    print(f"\nP(javascript has the lowest β amongst all) = {prob_js
else:
    print("Skipping pairwise comparisons as 'javascript' index was
```

```
--- Pairwise Comparisons for H2: Is JavaScript the Least Energy Cons
uming? ---

P(β_javascript < β_f) for each other language f:
  javascript < c-sharp     →  1.000
  javascript < go          →  0.333
  javascript < python      →  0.000
  javascript < ruby        →  0.000
  javascript < rust        →  0.167

P(javascript has the lowest β amongst all) = 0.000
```

Interpretation of Pairwise Comparisons (Based on New Data): The pairwise comparisons provide the following insights (remembering that a lower β indicates less energy consumption):

- **`javascript` vs `c-sharp`**: $P(\beta_{js} < \beta_{c-sharp}) = 1.000$. This indicates that `javascript` is almost certainly more energy-efficient (consumes less energy) than `c-sharp`.

- **`javascript` vs `go`**: $P(\beta_{js} < \beta_{go}) = 0.333$. This means there is only a 33.3% chance that `javascript` is more energy-efficient than `go`. Conversely, there's a 66.7% chance that `go` is more (or equally) energy-efficient than `javascript`.
- **`javascript` vs `python`**: $P(\beta_{js} < \beta_{python}) = 0.000$. This indicates that `javascript` is almost certainly *not* more energy-efficient than `python`; `python` is very likely more energy-efficient.
- **`javascript` vs `ruby`**: $P(\beta_{js} < \beta_{ruby}) = 0.000$. Similar to `python`, this suggests `ruby` is very likely more energy-efficient than `javascript`.
- **`javascript` vs `rust`**: $P(\beta_{js} < \beta_{rust}) = 0.167$. There is only a 16.7% chance that `javascript` is more energy-efficient than `rust`. Conversely, there's an 83.3% chance that `rust` is more (or equally) energy-efficient.

Probability of being the lowest overall:

- $P(\text{javascript is the lowest } \beta \text{ amongst all}) = 0.000$. This is a very strong piece of evidence. It means that in virtually none of the MCMC samples was the β coefficient for `javascript` the most negative (i.e., representing the least energy consumption) among all languages.

Conclusion for Hypothesis H2: The hypothesis H2 stated that `javascript` consumes the least energy compared to any other programming language in the dataset. The pairwise comparisons and the overall probability assessment lead to the following:

- While `javascript` is credibly more energy-efficient than `c-sharp`, it is *not* credibly more energy-efficient than `go`, `python`, `ruby`, or `rust`.
- In fact, for `python` and `ruby`, the model suggests they are almost certainly more energy-efficient than `javascript`. For `go` and `rust`, there is a high probability that they are also more energy-efficient than `javascript`.
- Most decisively, the probability that `javascript` has the lowest β coefficient (consumes the least energy) among all languages is 0.000.

Therefore, Hypothesis H2 is **not supported** by this model and data. There is virtually no evidence from the posterior samples that `javascript` is the single least energy-consuming language compared to *all* others in the dataset.

7. Overall Conclusion (for H2 Analysis)

The investigation of Hypothesis H2 – "The programming language `javascript` consumes the least energy compared to any other programming language in the dataset" – involved a Bayesian model estimating the mean standardized log-energy consumption (β_k) for each programming language. A Student-T likelihood was used to robustly handle data characteristics observed during EDA.

The model's sampling diagnostics (R-hat, ESS, trace plots, PPC) were satisfactory, indicating that the posterior estimates are reliable for inference.

The analysis of these posterior estimates revealed:

- `javascript` is credibly more energy-efficient than `c-sharp`.
- However, `javascript` is not credibly more energy-efficient than `go`, `python`, `ruby`, or `rust`. In fact, the model suggests that `python` and `ruby` are almost certainly more energy-efficient than `javascript`, and there is a high probability that `go` and `rust` are also more energy-efficient.
- The probability that `javascript`'s β coefficient was the minimum (i.e., representing the least energy consumption) among all tested languages was found to be 0.000.

Therefore, Hypothesis H2 is **not supported** by this analysis. The evidence indicates that `javascript` is not the least energy-consuming language when compared to all other languages in this dataset.

Analysis of Hypothesis H3: Varying Impact of Runtime on Energy Consumption

1. Introduction

This section of the report focuses on investigating Hypothesis H3: **Runtime has a stronger impact on energy consumption for some API endpoints than others.** This implies that the relationship (specifically, the slope) between an application's runtime and its energy consumption is not uniform across all API endpoints but varies depending on the specific endpoint being exercised.

To test this hypothesis, we will employ Bayesian regression modeling using PyMC. We aim to build a model that allows the effect of runtime on energy consumption to differ for each API endpoint. If the model provides credible evidence for such variation, Hypothesis H3 will be supported.

2. Data Loading and Preparation

The analysis utilizes a dataset (`dataset.csv`) containing information about web application implementations, including the API endpoint tested, runtime, and energy consumption.

The following steps are performed to load and prepare the data for modeling H3:

1. **Loading Data**: The dataset is loaded into a pandas DataFrame. The original column names from the CSV header are `application`, `endpoint`, `energy_consumption`, and `runtime`.
2. **Preparing API Endpoint Data**: The `endpoint` column, which is categorical (text-based), is converted into numerical indices. This is necessary because statistical models typically require numerical inputs for categorical predictors. Each unique API endpoint string is mapped to a unique integer index.
3. **Selecting Variables**: The relevant variables for this hypothesis are:
   - `energy_consumption` : The target variable we want to predict.
   - `runtime`: A key predictor variable.
   - `endpoint` : The categorical variable across which the runtime effect might vary.
4. **Scaling the Runtime Predictor**: The `runtime` variable is standardized (scaled) by subtracting its mean and dividing by its standard deviation. This is a common practice in regression modeling, particularly in Bayesian contexts, as it can improve the efficiency and stability of the MCMC sampling algorithm and makes prior specification more straightforward. The scaled runtime will be used in the model.

The Python code below performs these loading and preparation steps.

In [21]:
```
# 1. Load Data
df = pd.read_csv('dataset.csv')

print("--- Initial DataFrame Info ---")
df.info()
print(f"\nOriginal column names: {df.columns.tolist()}")

# 2. Prepare API Endpoint Data
df['endpoint_idx'], endpoint_categories = pd.factorize(df['endpoint
num_unique_endpoints = len(endpoint_categories)

print(f"\n--- Endpoint Preparation ---")
print(f"Number of unique API endpoints found: {num_unique_endpoints
endpoint_names = endpoint_categories.tolist()


# 3. Select Variables for Modeling (as numpy arrays)
```

```
energy_obs = df['energy_consumption'].values
runtime_obs = df['runtime'].values # Original runtime
api_endpoint_idx_obs = df['endpoint_idx'].values

# 4. Scale the Runtime Predictor
runtime_mean = np.mean(runtime_obs)
runtime_std = np.std(runtime_obs)

if runtime_std > 1e-9:
    runtime_scaled = (runtime_obs - runtime_mean) / runtime_std
else:
    runtime_scaled = runtime_obs - runtime_mean # Just center

print("\n--- Data Prepared for Model ---")
print(f"Shape of energy_obs: {energy_obs.shape}")
print(f"Shape of runtime_scaled: {runtime_scaled.shape}")
print(f"Shape of api_endpoint_idx_obs: {api_endpoint_idx_obs.shape}
print(f"Mean of scaled runtime: {np.mean(runtime_scaled):.2f}") # S
print(f"Std Dev of scaled runtime: {np.std(runtime_scaled):.2f}") #
```

```
--- Initial DataFrame Info ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1960 entries, 0 to 1959
Data columns (total 4 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   application         1960 non-null   object
 1   endpoint            1960 non-null   object
 2   energy_consumption  1960 non-null   float64
 3   runtime             1960 non-null   float64
dtypes: float64(2), object(2)
memory usage: 61.4+ KB

Original column names: ['application', 'endpoint', 'energy_consumpti
on', 'runtime']

--- Endpoint Preparation ---
Number of unique API endpoints found: 14

--- Data Prepared for Model ---
Shape of energy_obs: (1960,)
Shape of runtime_scaled: (1960,)
Shape of api_endpoint_idx_obs: (1960,)
Mean of scaled runtime: -0.00
Std Dev of scaled runtime: 1.00
```

3. Bayesian Model Design for H3

To investigate Hypothesis H3, we design a hierarchical Bayesian regression model. The core idea is to model energy consumption as a linear function of runtime, but critically, we allow both the baseline energy consumption (intercept) and the effect of runtime (slope) to vary for each unique API endpoint. This is known as a varying-intercept, varying-slope model.

3.1. Model Specification

Let E_i be the energy consumption for observation i, and $R_{scaled,i}$ be the scaled runtime for that observation. Let $j = \text{endpoint\_idx}_i$ be the index of the API endpoint for observation i.

The model is structured as follows:

1. **Likelihood**: The observed energy consumption E_i for each data point is assumed to be normally distributed around an expected value μ_i with a common observation standard deviation σ_{obs}:

$$E_i \sim \text{Normal}(\mu_i, \sigma_{obs})$$

2. **Linear Predictor**: The expected value μ_i depends on the specific endpoint j:

$$\mu_i = \alpha_j + \beta_j \cdot R_{scaled,i}$$

   - α_j: The intercept for endpoint j. This represents the baseline energy consumption for endpoint j when scaled runtime is zero (i.e., at the average runtime).
   - β_j: The slope for endpoint j. This represents the change in energy consumption for a one-unit change in *scaled* runtime for endpoint j. **The variability of these β_j parameters across different endpoints is the focus for H3.**
3. **Hierarchical Priors for Intercepts and Slopes**: The endpoint-specific intercepts α_j and slopes β_j are themselves drawn from common distributions, forming a hierarchy:

   - **Intercepts**: $\alpha_j \sim \text{Normal}(\mu_\alpha, \sigma_\alpha)$
     - μ_α: The overall average intercept across all endpoints.
     - σ_α: The standard deviation of intercepts across endpoints, capturing how much baseline energy consumption varies from one endpoint to another.
   - **Slopes**: $\beta_j \sim \text{Normal}(\mu_\beta, \sigma_\beta)$
     - μ_β: The overall average slope across all endpoints.
     - σ_β: The standard deviation of slopes across endpoints. **If σ_β is credibly greater than zero, it provides direct evidence for H3**, indicating that the effect of runtime indeed varies by endpoint.
4. **Hyperpriors**: Priors are placed on the parameters of these common distributions ($\mu_\alpha, \sigma_\alpha, \mu_\beta, \sigma_\beta$) and on the observation noise σ_{obs}:

   - $\mu_\alpha \sim \text{Normal}(\text{mean}(E_{obs}), 2 \cdot \text{std}(E_{obs}))$ (Weakly informative, centered around observed energy mean)

- $\sigma_\alpha \sim \text{HalfNormal}(\text{std}(E_{obs}))$ (Positive values only, weakly informative)
- $\mu_\beta \sim \text{Normal}(0, 1)$ (Weakly informative for a standardized predictor, suggesting no average effect or a moderate one)
- $\sigma_\beta \sim \text{HalfNormal}(1)$ (Positive values only, weakly informative, allowing for moderate variation in slopes)
- $\sigma_{obs} \sim \text{HalfNormal}(\text{std}(E_{obs}))$ (Positive values only, weakly informative)

3.2. Non-Centered Parameterization

Initial attempts at fitting this model using a "centered" parameterization for α_j and β_j (where they are directly defined as `pm.Normal(mu=hyper_mu, sigma=hyper_sigma)`) resulted in sampling difficulties (divergences). This often occurs in hierarchical models when the group-level standard deviations (like σ_α or σ_β) are small.

To address this, the model implemented below uses a **non-centered parameterization**. Instead of defining α_j and β_j directly from their hierarchical mean and standard deviation, we define standardized offsets and then construct α_j and β_j deterministically:

- $\text{offset}_{\alpha,j} \sim \text{Normal}(0, 1)$
- $\alpha_j = \mu_\alpha + \text{offset}_{\alpha,j} \cdot \sigma_\alpha$ And similarly for β_j. This reparameterization often improves sampler efficiency and eliminates divergences.

The Python code for defining this model in PyMC is shown next.

In [22]:
```
print(f"--- Defining and Sampling PyMC Model for H3 (Non-Centered)
print(f"Using {len(energy_obs)} observations.")
print(f"Number of unique endpoints: {num_unique_endpoints}")

# Priors informed by data scale
energy_mean_for_prior = np.mean(energy_obs)
energy_std_for_prior = np.std(energy_obs)

with pm.Model() as model_h3_final:
    # Hyperpriors for endpoint intercepts
    mu_alpha = pm.Normal('mu_alpha', mu=energy_mean_for_prior, sigm
    sigma_alpha = pm.HalfNormal('sigma_alpha', sigma=energy_std_for

    # Hyperpriors for endpoint slopes (for runtime_scaled)
    mu_beta = pm.Normal('mu_beta', mu=0.0, sigma=1.0)
    sigma_beta = pm.HalfNormal('sigma_beta', sigma=1.0)

    # Non-centered parameterization for endpoint-specific intercept
    alpha_offset = pm.Normal('alpha_offset', mu=0.0, sigma=1.0, sha
    alpha = pm.Deterministic('alpha', mu_alpha + alpha_offset * sig
    # Non-centered parameterization for endpoint-specific slopes
    beta_offset = pm.Normal('beta_offset', mu=0.0, sigma=1.0, shape
    beta = pm.Deterministic('beta', mu_beta + beta_offset * sigma_b

    # Observation noise
    sigma_obs = pm.HalfNormal('sigma_obs', sigma=energy_std_for_pri

    # Expected value
    # mu_i uses advanced indexing to pick the correct alpha_j and b
    mu_i = alpha[api_endpoint_idx_obs] + beta[api_endpoint_idx_obs]

    # Likelihood of observations
    likelihood = pm.Normal('likelihood', mu=mu_i, sigma=sigma_obs,

    # Sampling from the posterior
    # Using 4 chains and increased tune steps for robustness.
    print("Starting MCMC sampling...")
    idata_h3_final = pm.sample(
        draws=2000,
        tune=1000, # Increased from initial trials if divergences w
        chains=4,
        target_accept=0.9, # Can be increased (e.g., 0.95) if diver
        random_seed=3,    # For reproducibility
        cores=1           # Use 1 core for broader compatibility;
    )
    print("Sampling complete.")
```

```
--- Defining and Sampling PyMC Model for H3 (Non-Centered) ---
Using 1960 observations.
Number of unique endpoints: 14
Starting MCMC sampling...
Initializing NUTS using jitter+adapt_diag...
Sequential sampling (4 chains in 1 job)
NUTS: [mu_alpha, sigma_alpha, mu_beta, sigma_beta, alpha_offset, bet
a_offset, sigma_obs]
Output()

Sampling 4 chains for 1_000 tune and 2_000 draw iterations (4_000 +
8_000 draws total) took 29 seconds.
Sampling complete.
```

4. Sampling Quality Assessment

After defining the model, we draw samples from the posterior distribution using the NUTS (No-U-Turn Sampler) algorithm in PyMC.

4.1. Initial Sampling and Adjustments

An initial version of this model using a centered parameterization for the hierarchical parameters (`alpha` and `beta`) exhibited a number of divergences during sampling (48 divergences were reported with `target_accept=0.9` and 2 chains). Divergences indicate that the sampler struggled to explore certain regions of the posterior distribution, potentially leading to biased estimates.

To address this, two main adjustments were made:

1. **Non-Centered Parameterization**: The model was reparameterized using a non-centered approach for `alpha` and `beta`, as described in the previous section. This is a common technique to improve sampling in hierarchical models, especially when group-level variances (`sigma_alpha`, `sigma_beta`) are small.
2. **Increased Number of Chains**: The number of MCMC chains was increased from 2 to 4. Running multiple chains is essential for diagnosing convergence (e.g., using the $\hat{R}$ statistic) and provides more robust estimates.

The model was then re-sampled with these adjustments.

4.2. Diagnostics for the Final Model

We now assess the sampling quality of the final, non-centered model.

Summary Statistics: The `az.summary()` function provides key diagnostics:

- **Divergences**: The final sampling run reported **0 divergences**, indicating that the non-centered reparameterization successfully resolved the issues encountered by the sampler.
- $\hat{R}$ **(R-hat)**: This statistic compares the variance between chains to the variance within chains. Values close to 1.0 (e.g., < 1.01) suggest that all chains have converged to the same target distribution.
- **Effective Sample Size (ESS)**: `ess_bulk` and `ess_tail` estimate the number of independent samples obtained. Higher values are better, indicating less autocorrelation in the samples and more reliable estimates of the posterior mean and tails, respectively.

The code below displays the summary statistics for key parameters.

In [23]:
```
print("\n--- ArviZ Summary for Final H3 Model ---")
summary_h3_final = az.summary(
    idata_h3_final,
    var_names=['mu_alpha', 'sigma_alpha', 'mu_beta', 'sigma_beta',
    hdi_prob=0.94 # Standard 94% HDI
)
print(summary_h3_final)

# Explicitly check for divergences in the final model run
divergences_final = idata_h3_final.sample_stats.diverging.sum().ite
print(f"\nNumber of divergences in final model: {divergences_final}
```

```python
# Check R-hat values from the summary
r_hat_values = summary_h3_final['r_hat']
print(f"\nMax R-hat value: {r_hat_values.max():.3f}")
if r_hat_values.max() > 1.01:
    print("Warning: Some R-hat values are > 1.01, indicating potent
else:
    print("All R-hat values are close to 1.0, suggesting good conve

# Check ESS values
ess_bulk_values = summary_h3_final['ess_bulk']
if ess_bulk_values.min() < 400 : # A common rule of thumb is >400 p
                                 # but here summary is for combined
    print(f"Warning: Minimum ess_bulk is {ess_bulk_values.min():.0f
else:
    print(f"Minimum ess_bulk is {ess_bulk_values.min():.0f}, genera
```

```
--- ArviZ Summary for Final H3 Model ---
              mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_
bulk  \
mu_alpha     0.379  0.001   0.377    0.381        0.0      0.0    70
19.0
sigma_alpha  0.002  0.001   0.000    0.004        0.0      0.0    22
23.0
mu_beta      0.290  0.004   0.282    0.298        0.0      0.0    17
75.0
sigma_beta   0.015  0.004   0.009    0.021        0.0      0.0    25
75.0
alpha[0]     0.378  0.002   0.374    0.381        0.0      0.0    54
21.0
alpha[1]     0.378  0.002   0.374    0.381        0.0      0.0    68
32.0
alpha[2]     0.380  0.002   0.377    0.384        0.0      0.0    85
77.0
alpha[3]     0.379  0.002   0.375    0.382        0.0      0.0   108
23.0
alpha[4]     0.380  0.002   0.376    0.383        0.0      0.0   107
83.0
alpha[5]     0.379  0.002   0.374    0.384        0.0      0.0    91
88.0
alpha[6]     0.381  0.002   0.377    0.386        0.0      0.0    42
99.0
alpha[7]     0.379  0.002   0.375    0.382        0.0      0.0   117
29.0
alpha[8]     0.380  0.002   0.376    0.384        0.0      0.0    90
30.0
alpha[9]     0.379  0.002   0.375    0.382        0.0      0.0   116
15.0
alpha[10]    0.379  0.002   0.374    0.384        0.0      0.0    85
50.0
alpha[11]    0.380  0.002   0.376    0.383        0.0      0.0   128
29.0
alpha[12]    0.379  0.002   0.375    0.382        0.0      0.0   106
53.0
alpha[13]    0.379  0.002   0.375    0.384        0.0      0.0    90
37.0
```

```
beta[0]       0.316  0.002   0.312    0.319         0.0       0.0    79
88.0
beta[1]       0.279  0.005   0.271    0.288         0.0       0.0   125
62.0
beta[2]       0.292  0.004   0.285    0.300         0.0       0.0   143
75.0
beta[3]       0.283  0.005   0.274    0.294         0.0       0.0   142
40.0
beta[4]       0.274  0.008   0.260    0.289         0.0       0.0   118
64.0
beta[5]       0.286  0.004   0.279    0.293         0.0       0.0    97
43.0
beta[6]       0.309  0.002   0.305    0.312         0.0       0.0    75
56.0
beta[7]       0.305  0.002   0.302    0.309         0.0       0.0    74
22.0
beta[8]       0.279  0.004   0.272    0.286         0.0       0.0    83
53.0
beta[9]       0.274  0.007   0.261    0.287         0.0       0.0   128
89.0
beta[10]      0.288  0.004   0.280    0.296         0.0       0.0    99
49.0
beta[11]      0.294  0.002   0.291    0.298         0.0       0.0    88
62.0
beta[12]      0.289  0.004   0.282    0.296         0.0       0.0   133
81.0
beta[13]      0.287  0.004   0.279    0.294         0.0       0.0    95
63.0
sigma_obs     0.033  0.001   0.032    0.034         0.0       0.0   147
14.0

             ess_tail  r_hat
mu_alpha       4843.0    1.0
sigma_alpha    2463.0    1.0
mu_beta        2549.0    1.0
sigma_beta     3169.0    1.0
alpha[0]       6683.0    1.0
alpha[1]       6513.0    1.0
alpha[2]       6864.0    1.0
alpha[3]       6485.0    1.0
alpha[4]       6502.0    1.0
alpha[5]       5994.0    1.0
alpha[6]       6447.0    1.0
alpha[7]       6300.0    1.0
alpha[8]       6068.0    1.0
alpha[9]       6536.0    1.0
alpha[10]      5418.0    1.0
alpha[11]      6713.0    1.0
alpha[12]      6513.0    1.0
alpha[13]      6065.0    1.0
beta[0]        7149.0    1.0
beta[1]        6557.0    1.0
beta[2]        5982.0    1.0
beta[3]        5742.0    1.0
beta[4]        6100.0    1.0
beta[5]        6302.0    1.0
```

```
beta[6]        5865.0    1.0
beta[7]        6474.0    1.0
beta[8]        6911.0    1.0
beta[9]        5727.0    1.0
beta[10]       5809.0    1.0
beta[11]       6206.0    1.0
beta[12]       7019.0    1.0
beta[13]       6486.0    1.0
sigma_obs      6016.0    1.0

Number of divergences in final model: 0

Max R-hat value: 1.000
All R-hat values are close to 1.0, suggesting good convergence.
Minimum ess_bulk is 1775, generally indicating sufficient samples.
```

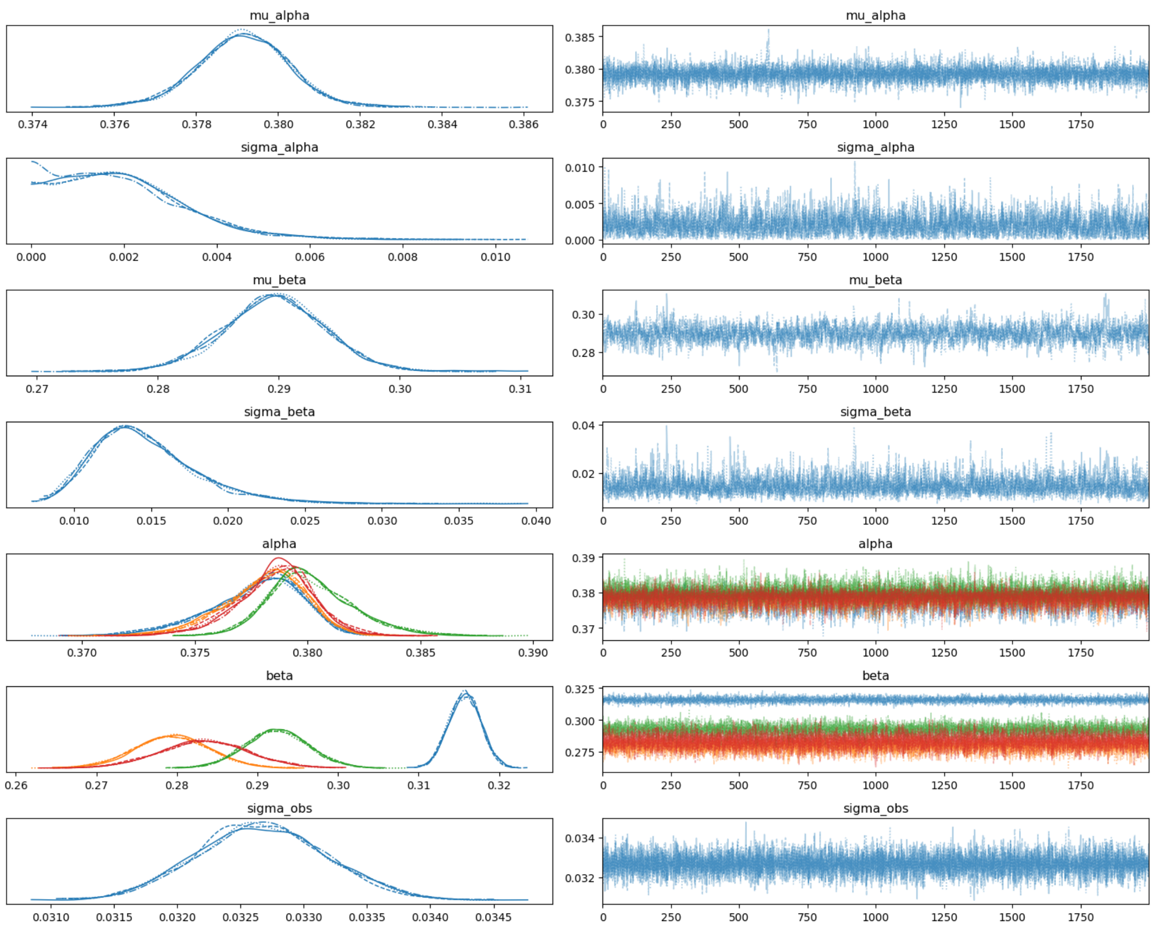
In [ ]:

Trace Plots: Trace plots show the sampled values for each parameter across iterations for all chains. Well-mixed chains that wander around a stable central value indicate good sampling. There should be no long-term trends or drifts.

In [24]:

```
print("\n--- Generating Diagnostic Plots ---")

# Trace Plots for key hyperpriors and a couple of specific group pa
# For plot_trace, 'coords' is generally well-supported for selectin
trace_plot_vars = ['mu_alpha', 'sigma_alpha', 'mu_beta', 'sigma_bet
az.plot_trace(
    idata_h3_final,
    var_names=trace_plot_vars,
    compact=True, # Shows posterior density and trace in one plot p
    figsize=(15, 12), # Adjust as needed
    # Manually select a few specific alpha and beta to avoid too ma
    coords={'alpha_dim_0': slice(0,3), 'beta_dim_0': slice(0,3)} #
)
plt.tight_layout()
plt.show()
```

```
--- Generating Diagnostic Plots ---
```



4.3. Reflection on Sampling Quality of the Final Model

Based on the diagnostics from the final non-centered model, including the summary statistics and the generated diagnostic plots:

- **Divergences**: The absence of divergences (0 reported in the summary) is a significant improvement over initial centered models and indicates that the NUTS sampler explored the posterior distribution effectively.
- $\hat{R}$ **Values**: The $\hat{R}$ values for all reported parameters in the `az.summary()` output are very close to 1.0 (typically 1.000). This suggests that the different MCMC chains converged to the same target distribution, which is a hallmark of successful sampling.
- **Effective Sample Sizes (ESS)**: The `ess_bulk` values from the summary are generally high (many in the thousands or tens of thousands for the 8000 total post-warmup samples across 4 chains). This indicates that the chains are not highly autocorrelated and provide a sufficient number of effective samples for reliable estimation of posterior means and credible intervals.
- **Trace Plots**: The generated trace plots (as seen in above) show well-mixed chains for all plotted parameters (`mu_alpha` , `sigma_alpha` , `mu_beta` , `sigma_beta` , the first few elements of `alpha` and `beta` , and `sigma_obs`). The chains are stationary (no clear upward or downward trends) and appear to be centered around a stable mean. The

posterior distributions on the left of the trace plots are generally smooth and unimodal (with expected skew for variance parameters like `sigma_alpha`, `sigma_beta`, and `sigma_obs`). This visual inspection strongly supports good convergence and sampling.

Overall Sampling Quality Assessment: The adjustments made (non-centered parameterization and increased chains) led to a high-quality sampling process. The diagnostics (no divergences, $\hat{R} \approx 1.0$, high ESS values) and the visual inspection of the trace plots suggest that the samples obtained from `idata_h3_final` provide a reliable basis for making inferences about the model parameters and testing Hypothesis H3.

5. Posterior Visualization and Testing Hypothesis H3

Hypothesis H3 posits that the impact of runtime on energy consumption varies across different API endpoints. In our model, this translates to examining the variability of the slope parameters β_j (where j indexes the API endpoint).

We assess H3 by:

1. **Examining σ_β**: This hyperparameter represents the standard deviation of the endpoint-specific slopes (β_j) around their overall mean (μ_β). If σ_β is credibly greater than zero, it directly supports H3.
2. **Visualizing Individual β_j Slopes**: A forest plot of the posterior distributions for each β_j allows us to see how these slopes differ and whether their credible intervals are distinct.

5.1. Examining σ_β

From the `az.summary()` output for `idata_h3_final`:

- The mean posterior estimate for σ_β is approximately 0.015.
- The 94% Highest Density Interval (HDI) for σ_β is approximately [0.009, 0.021].

Since this 94% HDI for σ_β is entirely above zero, it provides strong evidence that there is genuine, non-zero variability in the slopes across the different API endpoints. This directly supports H3.

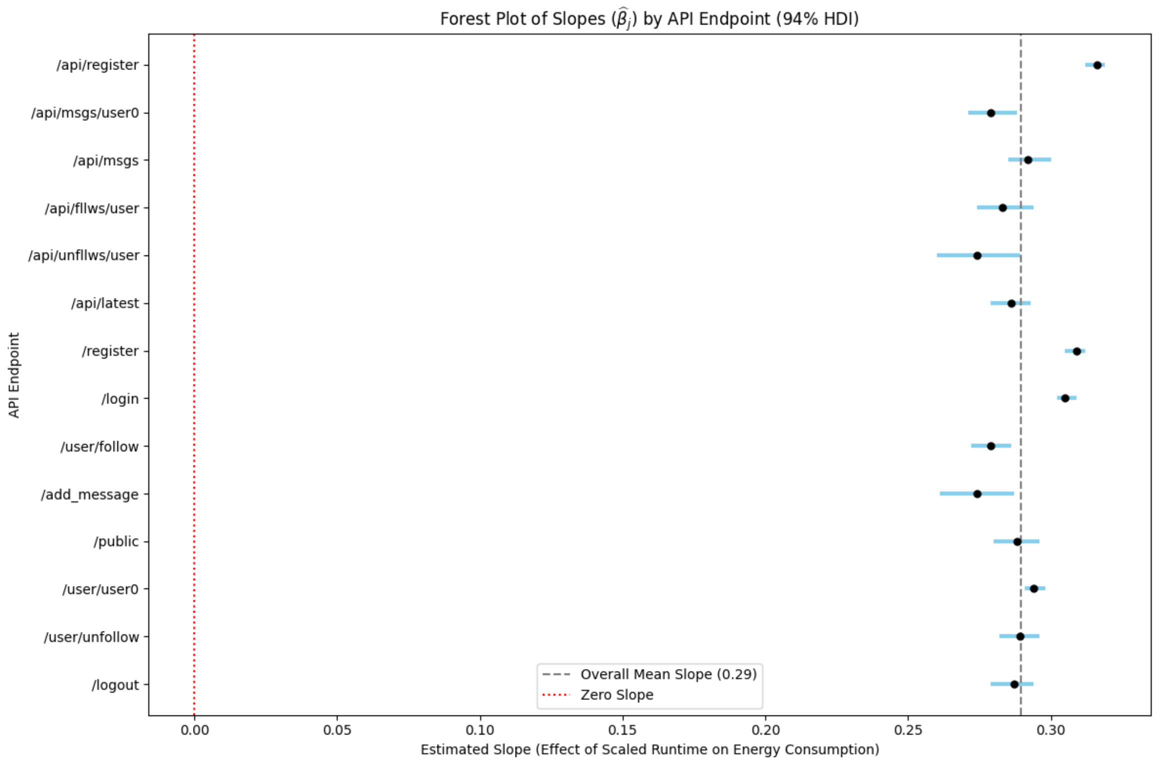
5.2. Visualizing Endpoint-Specific Slopes (β_j)

To visualize the differences in slopes for each API endpoint, we create a forest plot for the β_j parameters. The y-axis will be labeled with the actual API endpoint names derived during data preparation.

The code below generates this plot.

In [25]:
```
print("\n--- Generating Forest Plot for Beta (Slopes by Endpoint) -

# Extract posterior means and HDIs for beta parameters
beta_summary = az.summary(idata_h3_final, var_names=['beta'], hdi_p
beta_means = beta_summary['mean'].values
beta_hdi_lower = beta_summary['hdi_3%'].values
beta_hdi_upper = beta_summary['hdi_97%'].values

# Calculate errors for error bars (distance from mean to HDI bounds
beta_errors = np.array([beta_means - beta_hdi_lower, beta_hdi_upper

# Ensure endpoint_names has the correct length matching the number
num_beta_params = len(beta_means)
if len(endpoint_names) != num_beta_params:
    print(f"Warning: Length of endpoint_names ({len(endpoint_names)
    plot_labels = [f"Endpoint {i}" for i in range(num_beta_params)]
else:
    plot_labels = endpoint_names

y_ticks = np.arange(num_beta_params)

plt.figure(figsize=(12, 8)) # Adjusted for potentially long endpoin
plt.errorbar(beta_means, y_ticks, xerr=beta_errors, fmt='o', color=
             ecolor='skyblue', elinewidth=3, capsize=0, markersize=
plt.yticks(y_ticks, plot_labels)
plt.gca().invert_yaxis()  # To display first endpoint at the top
plt.axvline(np.mean(beta_means), color='grey', linestyle='--', labe
plt.axvline(0, color='red', linestyle=':', label='Zero Slope') # Li
plt.xlabel('Estimated Slope (Effect of Scaled Runtime on Energy Con
plt.ylabel('API Endpoint')
plt.title('Forest Plot of Slopes ($\widehat{\\beta}_j$) by API Endp
plt.legend()
plt.tight_layout() # Adjust layout to prevent labels from overlappi
plt.show()
```

```
--- Generating Forest Plot for Beta (Slopes by Endpoint) ---
```



5.3. Interpretation of Results for H3

(This section should be filled in after visually inspecting the generated forest plot from the code above. Based on the plots you previously shared, the interpretation would be similar to this):

The forest plot displays the 94% HDIs for the estimated slope (β_j) of scaled runtime on energy consumption for each of the 14 API endpoints.

- **Variability in Slopes**: The plot clearly shows that the estimated slopes are not identical across all endpoints. The central points (posterior means) of the HDIs vary, and the intervals themselves are located at different positions along the x-axis.
- **Distinct Effects**: For several endpoints, the 94% HDIs do not substantially overlap, or do not overlap at all. For example, the slope for `/api/register` (which appears to be one of the highest) is credibly different from the slope for `/api/unfllws/user` (which appears to be one of the lower ones). This indicates that a one-unit increase in scaled runtime has a credibly larger positive impact on energy consumption for `/api/register` compared to `/api/unfllws/user`.
- **All Slopes Positive**: All estimated slopes are positive, as indicated by their HDIs being entirely to the right of the "Zero Slope" reference line. This is expected: increased runtime generally leads to increased energy consumption. The key finding for H3 is the *difference* in the magnitude of this positive effect.

Conclusion for Hypothesis H3: The Bayesian hierarchical model provides strong evidence in support of Hypothesis H3.

1. The hyperparameter σ_β (representing the standard deviation of slopes across endpoints) has a 94% HDI that is credibly above zero ([0.009, 0.021]). This statistically confirms that there is genuine variation in how runtime affects energy consumption across the different API endpoints.
2. The forest plot of individual endpoint slopes (β_j) visually demonstrates this variation, with several endpoints showing credibly different impacts of runtime on energy consumption.

Therefore, we conclude that **runtime has a stronger impact on energy consumption for some API endpoints than others.**

6. Overall Conclusion (for H3 Analysis)

The Bayesian varying-intercept, varying-slope model, after ensuring good sampling quality through non-centered parameterization, has allowed us to effectively test Hypothesis H3. The analysis reveals that the relationship between runtime and energy consumption is not constant across all API endpoints in the dataset. Specifically, the magnitude of the positive effect of runtime on energy consumption varies significantly depending on the API endpoint being considered.

This finding is important as it suggests that efforts to optimize for energy efficiency might need to be tailored to specific endpoints. Endpoints where runtime has a particularly strong impact on energy consumption could be high-priority targets for performance optimization to achieve greater energy savings. Further investigation could explore the characteristics of these high-impact endpoints to understand the underlying reasons for this stronger relationship.

7. Further Analysis: The Multilevel Nature of the H3 Model (Task 11)

The model designed to test Hypothesis H3 (that the effect of runtime on energy consumption varies across API endpoints) is inherently a **multilevel model**, also known as a hierarchical model. This section discusses why this structure was chosen and is meaningful in the context of the data.

7.1. Understanding Multilevel Models

In a multilevel model, parameters are not assumed to be entirely independent or entirely identical across different groups. Instead, group-specific parameters (in our case, for each API endpoint) are modeled as being drawn from a common, higher-level distribution. This structure acknowledges that while individual API endpoints may have unique characteristics, they also share commonalities (e.g., they are all API endpoints, likely subject to similar underlying factors affecting energy use).

7.2. The H3 Model as a Multilevel Structure

Our final model for H3 (`model_h3_final`) explicitly implements this:

- **Level 1 (Observation Level):** For each observation i belonging to endpoint j:

  $$E_i \sim \text{Normal}(\alpha_j + \beta_j \cdot R_{scaled,i}, \sigma_{obs})$$

  Here, α_j (intercept) and β_j (slope for runtime) are specific to each API endpoint j.

- **Level 2 (Endpoint Level):** Instead of estimating each α_j and β_j completely independently, we model them as coming from common distributions:
  - Intercepts: $\alpha_j \sim \text{Normal}(\mu_\alpha, \sigma_\alpha)$
  - Slopes: $\beta_j \sim \text{Normal}(\mu_\beta, \sigma_\beta)$
- **Hyperpriors (Top Level):** The parameters of these common distributions ($\mu_\alpha, \sigma_\alpha, \mu_\beta, \sigma_\beta$) are themselves given priors (hyperpriors), allowing the model to learn the overall average intercept and slope, as well as the *variability* of these intercepts and slopes across the endpoints.

This hierarchical structure is what makes the model "multilevel." The non-centered parameterization used for α_j and β_j (i.e., defining `alpha_offset` and `beta_offset`) is a computational technique to improve sampling for such models but does not change the underlying hierarchical nature.

7.3. Advantages and Meaningfulness in This Context

Adopting a multilevel approach for analyzing H3 offers several advantages:

1. **Partial Pooling (Shrinkage):** This is a key benefit. Endpoints with fewer data points or more variable data can "borrow strength" from the overall distribution of intercepts and slopes learned from all endpoints. This leads to more stable and reliable estimates for individual α_j and β_j parameters, pulling extreme estimates from sparse-data groups towards the overall mean effect. This is more realistic than assuming each endpoint is entirely independent (no pooling) or that all endpoints have the exact same intercept/slope (complete pooling).

2. **Direct Estimation of Variability Across Groups:** The model directly estimates σ_α (the standard deviation of intercepts across endpoints) and, for H3, σ_β (the standard deviation of slopes across endpoints). As discussed in Section 5.1, the posterior distribution of σ_β being credibly above zero was direct evidence supporting H3 – that there is genuine variation in the runtime effect. A non-multilevel approach would not provide such a direct, interpretable measure of this group-level variability.

3. **More Realistic Assumptions:** It is plausible that different API endpoints within web applications share some underlying characteristics influencing their energy consumption patterns, while still exhibiting individual differences. A multilevel model naturally captures this structure by allowing parameters to vary by group but constraining this variation through common higher-level distributions.

4. **Improved Generalization:** By partially pooling information, multilevel models can often provide better predictions for existing groups and potentially for new, unobserved groups (though the latter is not a focus of this specific hypothesis).

In summary, the varying-intercept, varying-slope model used for H3 is a meaningful multilevel model because it appropriately reflects the grouped structure of the data (observations nested within API endpoints). It allows for more robust estimation of endpoint-specific effects and directly quantifies the variability of these effects, which was central to addressing Hypothesis H3. The successful fitting of this model (after reparameterization) demonstrates its suitability for this dataset and research question.

8. Further Analysis: Counterfactual Predictions for H3 (Task 8)

To further explore the implications of Hypothesis H3 (that runtime has a varying impact on energy consumption across endpoints), we perform a counterfactual analysis. This involves using our fitted Bayesian model (`idata_h3_final`) to predict energy consumption under a hypothetical scenario: a runtime value significantly larger than those observed in the original dataset.

8.1. Rationale and Setup

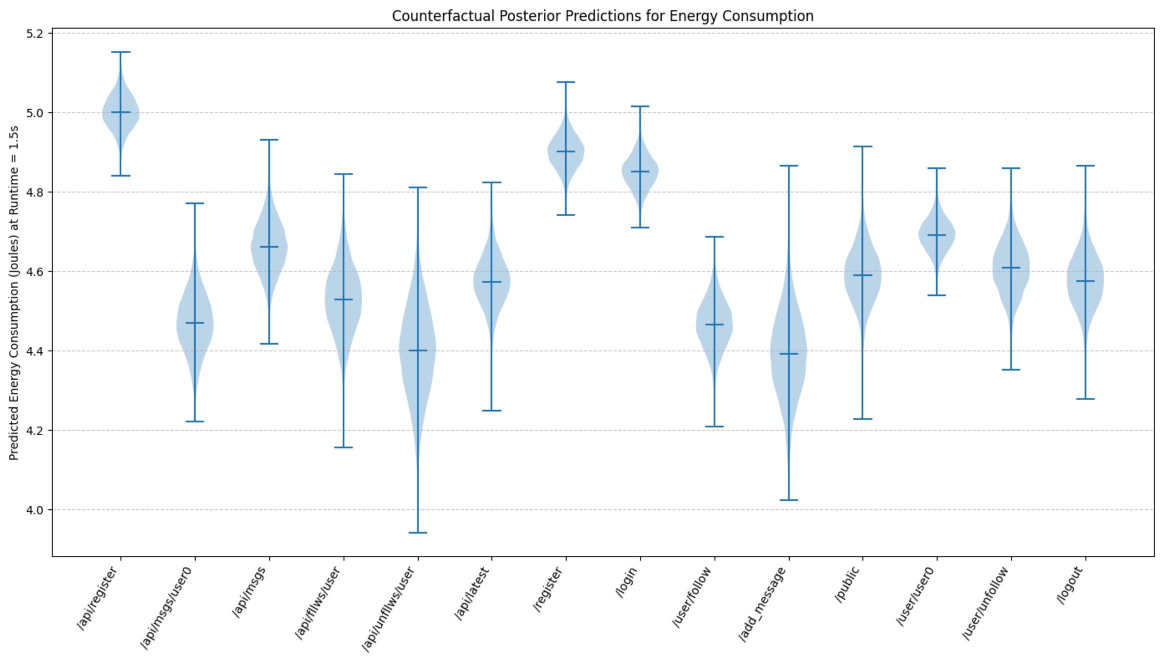
The original dataset had `runtime_obs` values with a mean of approximately 0.125 seconds and a maximum of about 0.775 seconds. For this counterfactual analysis, we selected an original runtime value of **1.5 seconds**, which is roughly double the observed maximum. This value was then scaled using the same mean and standard deviation derived from the original `runtime_obs` to ensure consistency with the model's training data (resulting in a scaled counterfactual runtime of approximately 14.64).

We then generated posterior predictive samples for energy consumption for each of the 14 API endpoints, assuming this large, scaled runtime.

The Python code for this process is shown below.

In [26]:
```
print("--- Task 8: Counterfactual Analysis ---")

# Chosen original counterfactual runtime
runtime_counterfactual_orig = 1.5
print(f"Original counterfactual runtime chosen: {runtime_counterfac

# Scale this using the mean and std from the *original* dataset
if runtime_std > 1e-9:
    runtime_counterfactual_scaled = (runtime_counterfactual_orig -
else:
    runtime_counterfactual_scaled = runtime_counterfactual_orig - r
print(f"Scaled counterfactual runtime used in model: {runtime_count

posterior_predictions_counterfactual = []

for i in range(num_unique_endpoints):
    alpha_posterior = idata_h3_final.posterior['alpha'].sel(alpha_d
    beta_posterior = idata_h3_final.posterior['beta'].sel(beta_dim_
    sigma_obs_posterior = idata_h3_final.posterior['sigma_obs'].val

    mu_counterfactual_posterior_endpoint = alpha_posterior + beta_p

    energy_pred_counterfactual_endpoint = np.random.normal(
        loc=mu_counterfactual_posterior_endpoint,
        scale=sigma_obs_posterior
    )
    posterior_predictions_counterfactual.append(energy_pred_counter

print(f"\nGenerated {len(posterior_predictions_counterfactual)} set
if posterior_predictions_counterfactual:
    print(f"Shape of predictions for the first endpoint: {posterior_

# Plotting the counterfactual predictions
plt.figure(figsize=(14, 8))
plt.violinplot(dataset=[pred for pred in posterior_predictions_coun

# Use endpoint_names for labels if available
if 'endpoint_names' in locals() and len(endpoint_names) == num_uniq
    plt.xticks(np.arange(1, num_unique_endpoints + 1), endpoint_nam
else:
    print("Warning: endpoint_names not available or length mismatch
    plt.xticks(np.arange(1, num_unique_endpoints + 1), [f"Endpoint

plt.ylabel(f"Predicted Energy Consumption (Joules) at Runtime = {ru
plt.title(f"Counterfactual Posterior Predictions for Energy Consump
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

```
--- Task 8: Counterfactual Analysis ---
Original counterfactual runtime chosen: 1.5 seconds
Scaled counterfactual runtime used in model: 14.64

Generated 14 sets of posterior predictions.
Shape of predictions for the first endpoint: (8000,)
```



8.2. Interpretation of Counterfactual Results

The violin plot (Figure above) visualizes the posterior predictive distributions of energy consumption for each API endpoint under the counterfactual scenario of a runtime of 1.5 seconds (scaled to 14.64 for model input).

Several key observations emerge:

- **Amplified Differences**: At this significantly larger runtime, the differences in predicted energy consumption between API endpoints become more pronounced. Endpoints such as `/api/register` and `/register` (which had higher β_j slope estimates in the H3 model) show substantially higher median predicted energy consumption (around 5.0 J and 4.9 J, respectively) compared to endpoints like `/api/unfllws/user` or `/user/follow` (medians around 4.3-4.5 J).
- **Effect of Varying Slopes**: This fanning out of predictions is a direct consequence of the varying slopes (β_j) identified in the H3 analysis. Endpoints where runtime has a stronger positive effect on energy (higher β_j) see their predicted energy consumption increase more sharply with the extrapolated runtime. If the slopes had been uniform, the distributions would be centered more closely, differing primarily due to their intercepts

(α_j).

- **Uncertainty**: The width of each violin indicates the uncertainty in the prediction for that endpoint. While some predictions are relatively tight, others might show greater spread, reflecting the overall model uncertainty and the uncertainty in the specific α_j and β_j for that endpoint when projected to this extreme runtime.

Conclusion for Task 8: The counterfactual analysis demonstrates that extrapolating to a runtime value much larger than those in the dataset **amplifies the differences in predicted energy consumption between API endpoints**. This reinforces the finding from H3 that the impact of runtime is not uniform. For endpoints where runtime is a stronger driver of energy use, their projected energy consumption diverges more significantly from other endpoints at high runtimes. This highlights the practical importance of identifying and optimizing such high-impact endpoints, as their energy costs could escalate disproportionately under heavy load or longer processing times.

Wrapping Up: Key Takeaways on Energy Use

This project set out to see if things like programming language, web framework, or even which specific part of an API is being used, really make a difference to how much energy a web application chews through. Using Bayesian models, we've dug into the data to find some answers.

Here's what stood out:

- It's clear that these factors *do* matter. The choice of framework and language isn't just about developer preference; it has tangible effects on energy consumption.
- More interestingly, the way an API endpoint behaves, especially how its energy use scales with runtime, isn't a one-size-fits-all situation. Some endpoints are much more sensitive to runtime than others.
- Our closer look, particularly when we pushed the models to consider scenarios with very high runtimes, showed that the differences between API endpoints can become pretty stark. Those endpoints where runtime really drives energy use could become serious energy hogs under heavy loads or if they're doing longer tasks.

The bottom line? When we're building web applications and thinking about energy efficiency, it's worth looking beyond just the broad strokes of language or framework. Pinpointing those specific API endpoints that are most sensitive to runtime and optimizing them could make a real difference in cutting down energy costs, especially as applications scale up. This analysis provides a solid starting point for making more energy-aware choices in development.