Problem 1: Bias-Variance Tradeoff and Regularization

In this problem, we will explore the fundamental concepts of the bias-variance tradeoff and demonstrate how regularization affects model performance.

Part 1: Deriving the Bias-Variance Decomposition [15 points]

Task: Show mathematically that the expected test error can be decomposed as:

$$E[(y - \hat{f}(x))^2] = \text{Bias}^2 + \text{Variance} + \text{Irreducible Noise}$$

where:

- $y = f(x) + \epsilon$ is the true target with noise $\epsilon \sim N(0, \sigma^2)$
- $\hat{f}(x)$ is our estimated function
- $\text{Bias} = E[\hat{f}(x)] - f(x)$
- $\text{Variance} = E[(\hat{f}(x) - E[\hat{f}(x)])^2]$

Derivation

Let x be a fixed test point. Assume the data-generating model

$$y = f(x) + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2),$$

and $\hat{f}(x)$ is a predictor learned from a random training dataset D. We derive a decomposition of the expected squared test error at x: $\mathbb{E}\big[(y - \hat{f}(x))^2\big]$.

$$\begin{aligned}
\mathbb{E}\big[(y - \hat{f}(x))^2\big] &= \mathbb{E}\Big[\big(f(x) + \varepsilon - \hat{f}(x)\big)^2\Big] \\
&= \mathbb{E}\Big[\big((f(x) - \hat{f}(x)) + \varepsilon\big)^2\Big] \\
&= \mathbb{E}\Big[(f(x) - \hat{f}(x))^2 + 2\varepsilon(f(x) - \hat{f}(x)) + \varepsilon^2\Big] \\
&= \mathbb{E}\big[(f(x) - \hat{f}(x))^2\big] + 2\mathbb{E}\big[\varepsilon(f(x) - \hat{f}(x))\big] + \mathbb{E}[\varepsilon^2].
\end{aligned}$$

We handle the cross term using iterated expectation w.r.t. the training data D (equivalently w.r.t. $\hat{f}(x)$) and the independence of ε from D:

$$\begin{aligned}\mathbb{E}\big[\varepsilon(f(x)-\hat{f}\,(x))\big] &= \mathbb{E}_D\Big[\mathbb{E}_\varepsilon\big[\varepsilon(f(x)-\hat{f}\,(x)) \mid D\big]\Big] \\ &= \mathbb{E}_D\Big[(f(x)-\hat{f}\,(x))\,\mathbb{E}_\varepsilon[\varepsilon \mid D]\Big] \\ &= \mathbb{E}_D\Big[(f(x)-\hat{f}\,(x))\,\mathbb{E}_\varepsilon[\varepsilon]\Big] \\ &= \mathbb{E}_D\Big[(f(x)-\hat{f}\,(x))\cdot 0\Big] \\ &= 0.\end{aligned}$$

Also,

$$\mathbb{E}[\varepsilon^2] = \mathrm{Var}(\varepsilon) + (\mathbb{E}[\varepsilon])^2 = \sigma^2 + 0 = \sigma^2.$$

Hence

$$\mathbb{E}\big[(y-\hat{f}\,(x))^2\big] = \mathbb{E}\big[(f(x)-\hat{f}\,(x))^2\big] + \sigma^2.$$

Now define the mean prediction (over training sets)

$$\mu(x) := \mathbb{E}[\hat{f}\,(x)].$$

Add and subtract $\mu(x)$ inside the square:

$$\begin{aligned}\mathbb{E}\big[(f(x)-\hat{f}\,(x))^2\big] &= \mathbb{E}\Big[\big(f(x)-\mu(x)+\mu(x)-\hat{f}\,(x)\big)^2\Big] \\ &= \mathbb{E}\Big[(f(x)-\mu(x))^2 + 2(f(x)-\mu(x))(\mu(x)-\hat{f}\,(x)) + (\mu(\\ &= (f(x)-\mu(x))^2 + 2(f(x)-\mu(x))\,\mathbb{E}[\mu(x)-\hat{f}\,(x)] + \mathbb{E}[(\mu \\ &= (f(x)-\mu(x))^2 + 2(f(x)-\mu(x))\big(\mu(x)-\mathbb{E}[\hat{f}\,(x)]\big) + \mathbb{E}[\\ &= (f(x)-\mu(x))^2 + 2(f(x)-\mu(x))(\mu(x)-\mu(x)) + \mathbb{E}[(\hat{f}\,(\\ &= (f(x)-\mu(x))^2 + \mathbb{E}[(\hat{f}\,(x)-\mu(x))^2].\end{aligned}$$

Using the given definitions,

$$\mathrm{Bias}(x) = \mathbb{E}[\hat{f}\,(x)] - f(x) = \mu(x) - f(x) \quad \Rightarrow \quad (f(x)-\mu(x))^2 = (\mu(x)-f($$

and

$$\mathrm{Var}(x) = \mathbb{E}\Big[(\hat{f}\,(x)-\mathbb{E}[\hat{f}\,(x)])^2\Big] = \mathbb{E}[(\hat{f}\,(x)-\mu(x))^2].$$

Therefore,

$$\mathbb{E}\big[(f(x)-\hat{f}\,(x))^2\big] = \mathrm{Bias}(x)^2 + \mathrm{Var}(x).$$

Substituting back,

$$\mathbb{E}\big[(y - \hat{f}(x))^2\big] = \Big(\mathrm{Bias}(x)^2 + \mathrm{Var}(x)\Big) + \sigma^2$$
$$= \mathrm{Bias}(x)^2 + \mathrm{Var}(x) + \sigma^2.$$

Thus,

$$\mathbb{E}\big[(y - \hat{f}(x))^2\big] = \mathrm{Bias}(x)^2 + \mathrm{Variance}(x) + \mathrm{Irreducible\ Noise}$$

Part 2: Dataset Creation and Visualization [10 points]

Task: Create a dataset using the following specifications:

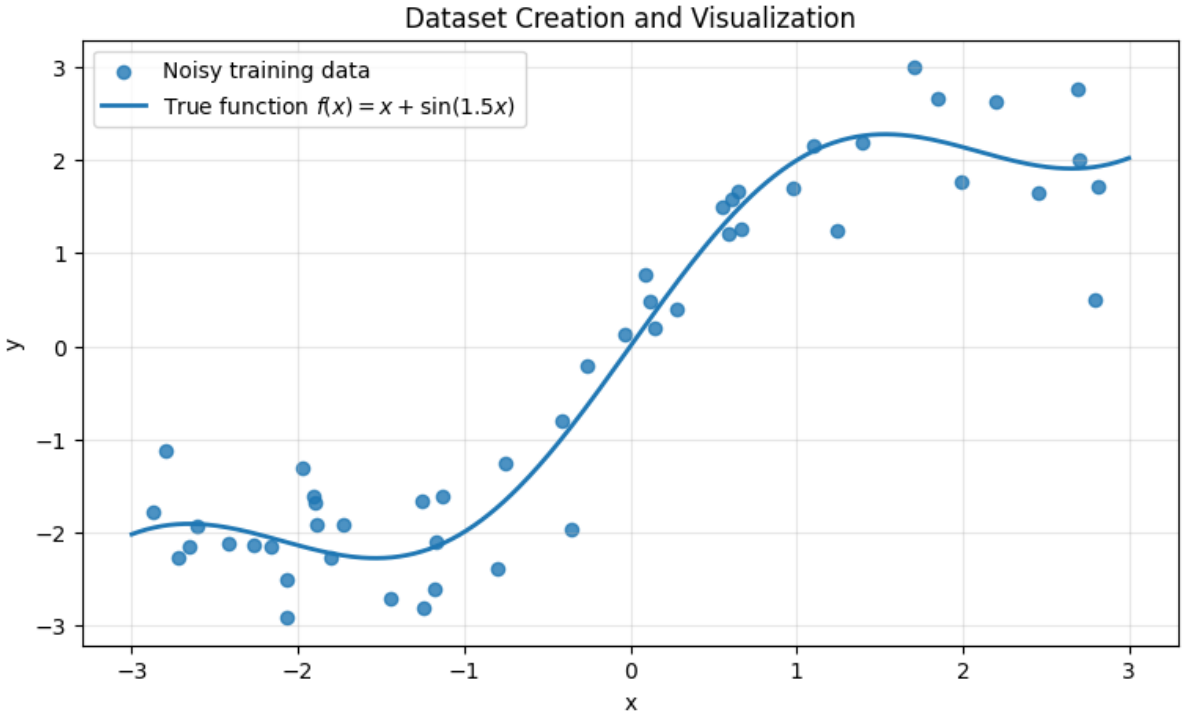
- True function: $f(x) = x + \sin(1.5x)$
- Add Gaussian noise with variance $\sigma^2 = 0.3$
- Generate 50 training samples with $x \in [-3, 3]$

Then create a visualization showing:

1. The noisy training data points
2. The true underlying function $f(x)$

In [10]:
```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(42)

# TODO: Define the true function f(x)
def f(x):
    return x + np.sin(1.5 * x)

# TODO: Define function y(x) that adds noise to f(x)
def y(x, noise_var):
    noise_std = np.sqrt(noise_var)
    return f(x) + np.random.normal(loc=0.0, scale=noise_std, size=x.sh

# TODO: Set parameters
n_samples = 50  # number of training samples
x_range = (-3, 3)    # tuple (min, max) for x values
noise_var = 0.3  # noise variance σ²

# TODO: Generate training data
X_train = np.random.uniform(x_range[0], x_range[1], n_samples)
y_train = y(X_train, noise_var)

# TODO: Create visualization
x_plot = np.linspace(x_range[0], x_range[1], 400)
```

```
y_true = f(x_plot)

plt.figure(figsize=(9, 5))
plt.scatter(X_train, y_train, s=35, alpha=0.8, label="Noisy training d
plt.plot(x_plot, y_true, linewidth=2, label=r"True function $f(x)=x+\s
plt.xlabel("x")
plt.ylabel("y")
plt.title("Dataset Creation and Visualization")
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
```



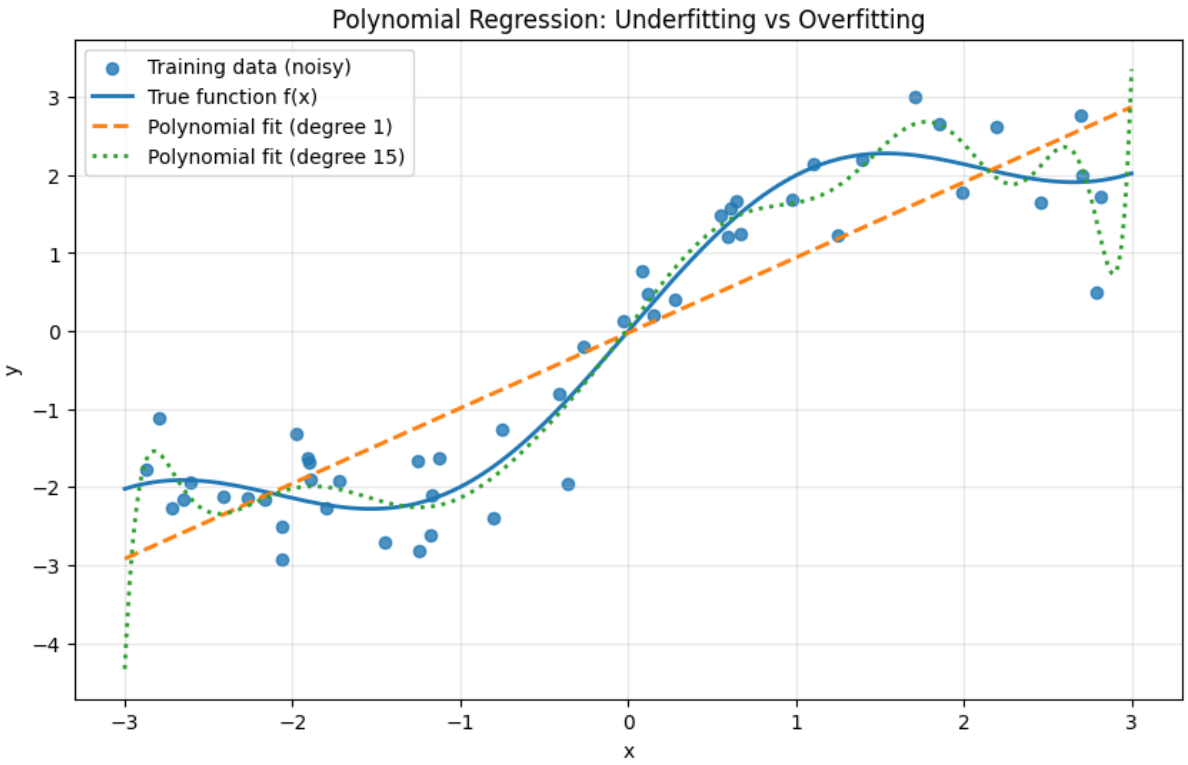
Part 3: Polynomial Fitting - Underfitting vs Overfitting [15 points]

Task: Demonstrate underfitting and overfitting using polynomial regression:

1. Fit a **low-degree polynomial** (e.g., degree 1) to show underfitting
2. Fit a **high-degree polynomial** (e.g., degree 15) to show overfitting
3. Visualize both fits along with the true function and training data

In [11]:
```
# TODO: Fit polynomials of different degrees
low_degree = 1   # underfitting
high_degree = 15  # overfitting

# TODO: Fit the polynomials using np.polyfit
coef_low = np.polyfit(X_train, y_train, deg=low_degree)
```

```
coef_high = np.polyfit(X_train, y_train, deg=high_degree)

poly_low = np.poly1d(coef_low)
poly_high = np.poly1d(coef_high)

x_plot = np.linspace(x_range[0], x_range[1], 600)

# Predictions
y_true = f(x_plot)
y_low = poly_low(x_plot)
y_high = poly_high(x_plot)

# TODO: Create visualization comparing underfitting vs overfitting
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, s=35, alpha=0.8, label="Training data (n
plt.plot(x_plot, y_true, linewidth=2, label="True function f(x)")
plt.plot(x_plot, y_low, linewidth=2, linestyle="--", label=f"Polynomia
plt.plot(x_plot, y_high, linewidth=2, linestyle=":", label=f"Polynomia

plt.xlabel("x")
plt.ylabel("y")
plt.title("Polynomial Regression: Underfitting vs Overfitting")
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
```



Part 4: Bias-Variance Tradeoff Analysis [15

points]

Task: Empirically demonstrate the bias-variance tradeoff:

1. For polynomial degrees 1 to 15, fit models on multiple different datasets (at least 100 datasets)
2. Compute the Bias², Variance, and MSE for each degree
3. Create a plot showing how these quantities change with model complexity
4. Identify the optimal model complexity

In [12]:
```
# TODO: Set up parameters for analysis
n_datasets = 100  # number of different datasets to generate
max_degree = 15   # maximum polynomial degree to test
n_test = 200      # number of test points

# TODO: Generate fixed test points
X_test = np.linspace(x_range[0], x_range[1], n_test)
y_test_true = f(X_test)

# Store predictions: shape (max_degree, n_datasets, n_test)
preds = np.zeros((max_degree, n_datasets, n_test))

# TODO: For each dataset, fit polynomials of each degree and store pre
rng = np.random.default_rng(42)

for d in range(n_datasets):
    # Generate a fresh noisy training dataset
    X_train = rng.uniform(x_range[0], x_range[1], n_samples)
    y_train = f(X_train) + rng.normal(0.0, np.sqrt(noise_var), size=X_

    for deg in range(1, max_degree + 1):
        coefs = np.polyfit(X_train, y_train, deg=deg)
        model = np.poly1d(coefs)
        preds[deg - 1, d, :] = model(X_test)
```

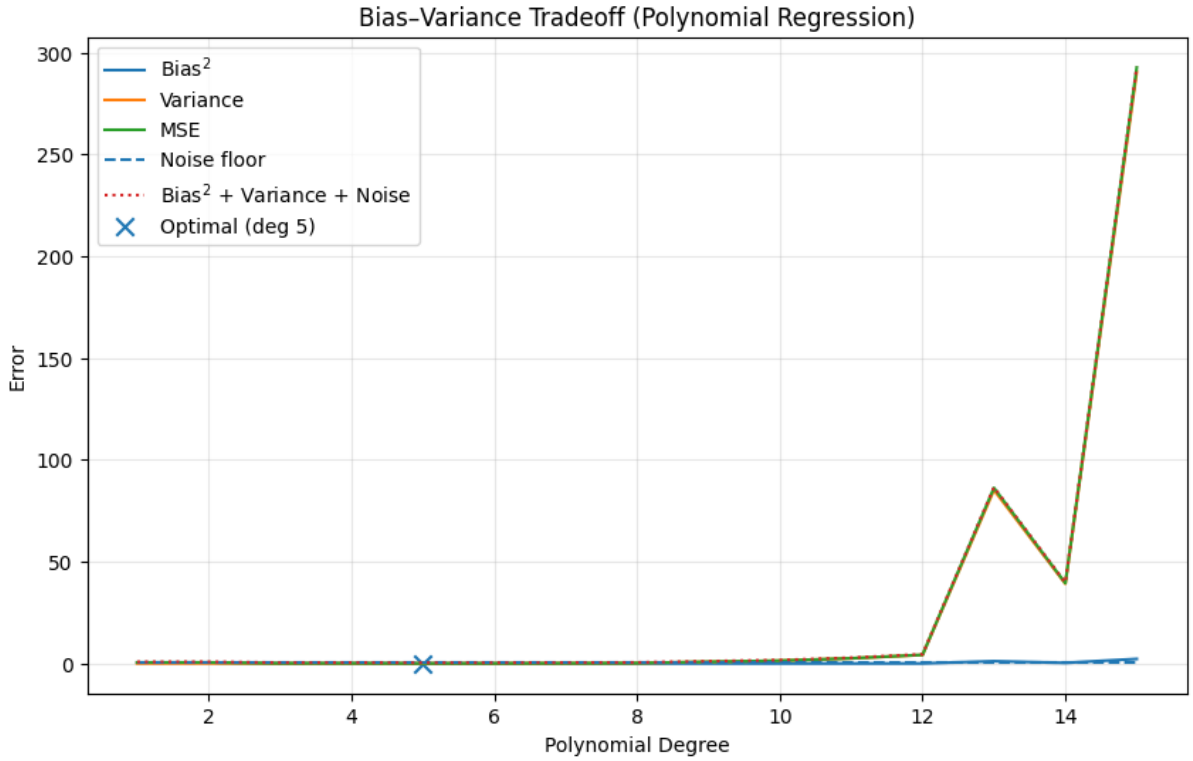
In [13]:
```
# TODO: Compute Bias², Variance, and MSE for each polynomial degree
bias2 = np.zeros(max_degree)
var = np.zeros(max_degree)
mse = np.zeros(max_degree)

for deg in range(1, max_degree + 1):
    Y_hat = preds[deg - 1]                 # (n_datasets, n_test)
    mean_pred = np.mean(Y_hat, axis=0)     # (n_test,)

    bias2[deg - 1] = np.mean((mean_pred - y_test_true) ** 2)
    var[deg - 1] = np.mean(np.var(Y_hat, axis=0, ddof=0))
    mse[deg - 1] = np.mean((Y_hat - y_test_true) ** 2)  # E[(f_hat - f

# Theoretical noise floor (irreducible error)
noise_floor = noise_var
```

```
decomp = bias2 + var + noise_floor

degrees = np.arange(1, max_degree + 1)
opt_deg = degrees[np.argmin(mse)]

# TODO: Print results in a table format
print(f"{'Degree':>6} | {'Bias^2':>10} | {'Variance':>10} | {'MSE':>10
print("-" * 65)
for i, deg in enumerate(degrees):
    print(f"{deg:6d} | {bias2[i]:10.5f} | {var[i]:10.5f} | {mse[i]:10.

print(f"\nOptimal degree (min MSE): {opt_deg}")
```

```
Degree |     Bias^2 |   Variance |        MSE |  Bias^2+Var+Noise
-----------------------------------------------------------------
     1 |    0.47947 |    0.03135 |    0.51081 |            0.81081
     2 |    0.47961 |    0.05172 |    0.53133 |            0.83133
     3 |    0.07068 |    0.03882 |    0.10951 |            0.40951
     4 |    0.07153 |    0.06110 |    0.13263 |            0.43263
     5 |    0.00190 |    0.05442 |    0.05633 |            0.35633
     6 |    0.00197 |    0.07229 |    0.07426 |            0.37426
     7 |    0.00239 |    0.12170 |    0.12409 |            0.42409
     8 |    0.00183 |    0.21219 |    0.21402 |            0.51402
     9 |    0.00182 |    0.80723 |    0.80905 |            1.10905
    10 |    0.01183 |    1.35876 |    1.37058 |            1.67058
    11 |    0.01201 |    2.51435 |    2.52636 |            2.82636
    12 |    0.03115 |    4.21165 |    4.24280 |            4.54280
    13 |    1.13111 |   85.13001 |   86.26112 |           86.56112
    14 |    0.31219 |   39.15747 |   39.46966 |           39.76966
    15 |    2.24087 |  290.45243 |  292.69330 |          292.99330

Optimal degree (min MSE): 5
```

In [14]:

```
# TODO: Create the bias-variance tradeoff plot
# Show: Bias², Variance, MSE, Noise floor, and Bias²+Variance+Noise
# Mark the optimal model (minimum MSE)
plt.figure(figsize=(10, 6))
plt.plot(degrees, bias2, label=r"Bias$^2$")
plt.plot(degrees, var, label="Variance")
plt.plot(degrees, mse, label="MSE")
plt.hlines(noise_floor, xmin=1, xmax=max_degree, linestyles="--", labe
plt.plot(degrees, decomp, linestyle=":", label=r"Bias$^2$ + Variance +

# Mark the optimal model
plt.scatter([opt_deg], [mse[opt_deg - 1]], s=80, marker="x", label=f"O

plt.xlabel("Polynomial Degree")
plt.ylabel("Error")
plt.title("Bias–Variance Tradeoff (Polynomial Regression)")
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
```



Part 4: Analysis - Best Model Identification

Can you identify the best model?

The best model is the polynomial of degree 5, because it achieves the lowest test MSE in the bias-variance analysis. For degrees lower than 5, the model underfits (high bias), while for degrees higher than 5 the variance grows rapidly (overfitting), which increases the MSE. Degree 5 provides the best balance between bias and variance on this dataset.

Part 5: L2 Regularization (Ridge Regression) [10 points]

Task: Apply L2 regularization to the degree-10 polynomial and compare with the unregularized version.

In [15]:
```
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures

# TODO: Set parameters
degree = 10
alpha = 1.0  # Regularization strength (lambda)
```

```
n_datasets = 100
n_test = 200

# Fixed test set
X_test = np.linspace(x_range[0], x_range[1], n_test)
y_test_true = f(X_test)

# Precompute polynomial feature transformer
poly = PolynomialFeatures(degree=degree, include_bias=True)

# Store predictions across datasets
preds_unreg = np.zeros((n_datasets, n_test))
preds_ridge = np.zeros((n_datasets, n_test))

rng = np.random.default_rng(42)

# TODO: Fit unregularized and regularized polynomials on multiple data
for d in range(n_datasets):
    X_train = rng.uniform(x_range[0], x_range[1], n_samples)
    y_train = f(X_train) + rng.normal(0.0, np.sqrt(noise_var), size=X_

    # Shape for sklearn: (n, 1)
    Xtr = X_train.reshape(-1, 1)
    Xte = X_test.reshape(-1, 1)

    Phi_tr = poly.fit_transform(Xtr)
    Phi_te = poly.transform(Xte)

    # Unregularized degree-10 polynomial via least squares on polynomi
    w_unreg, *_ = np.linalg.lstsq(Phi_tr, y_train, rcond=None)
    preds_unreg[d, :] = Phi_te @ w_unreg

    # Ridge-regularized polynomial
    ridge = Ridge(alpha=alpha, fit_intercept=False)  # intercept alrea
    ridge.fit(Phi_tr, y_train)
    preds_ridge[d, :] = ridge.predict(Phi_te)
```

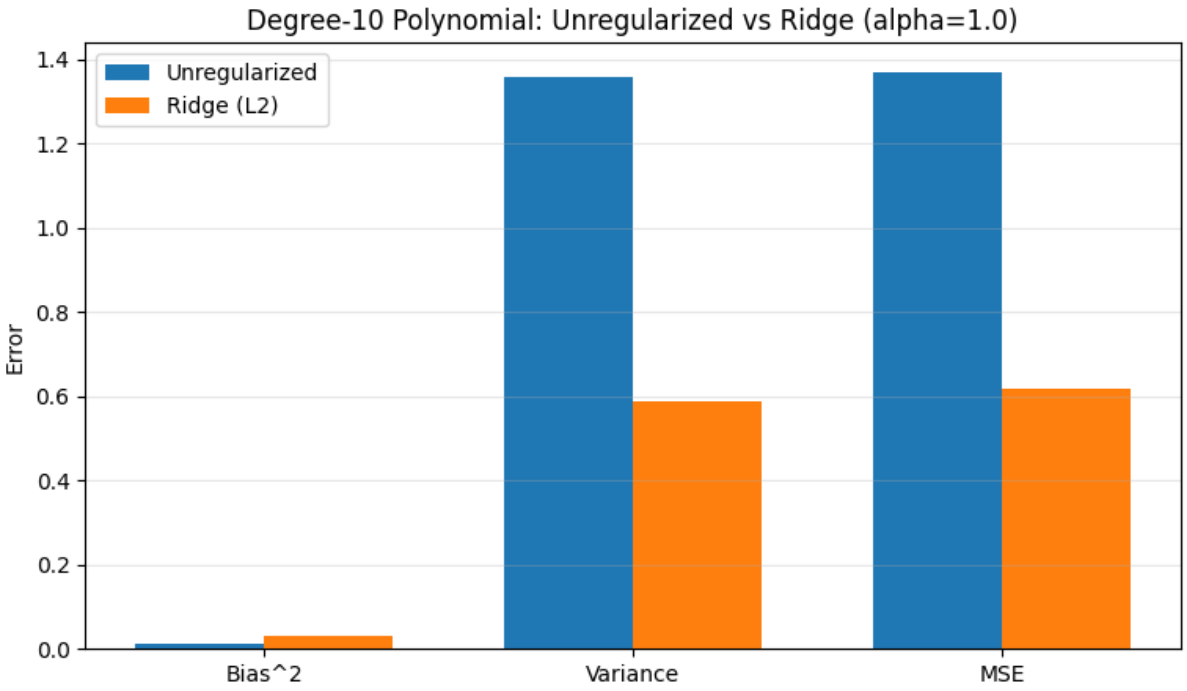
In [16]:

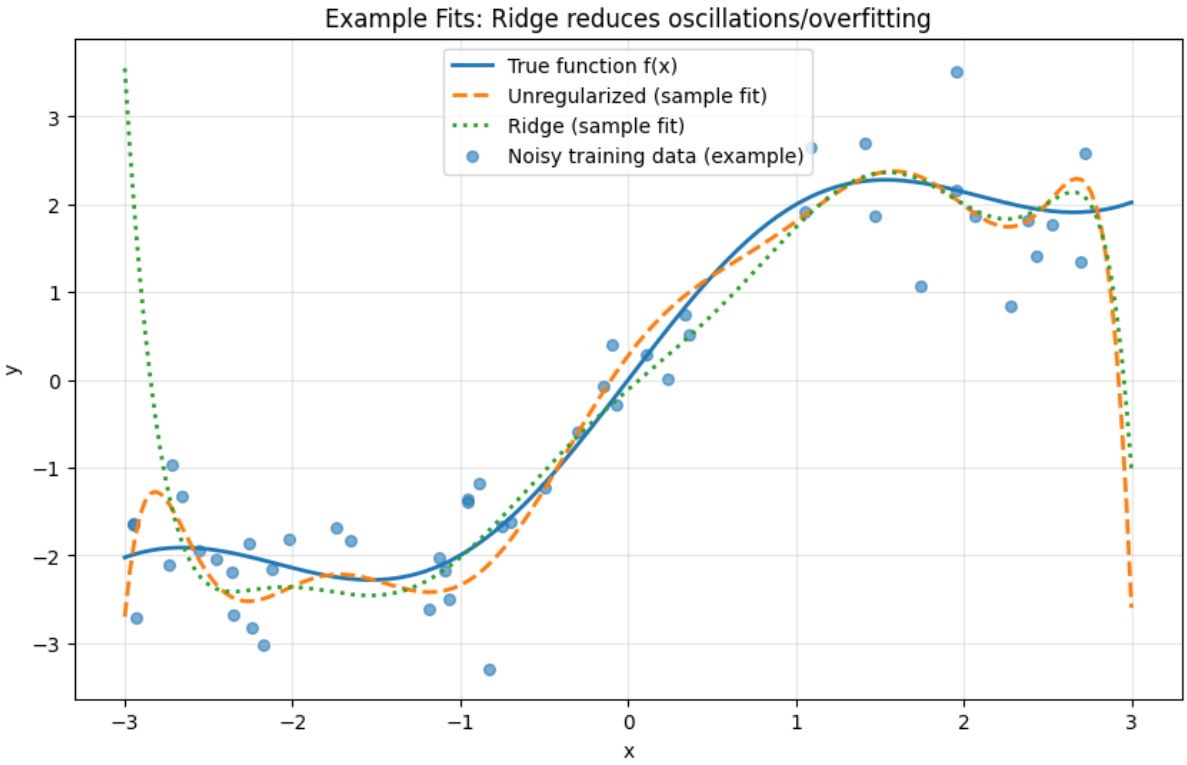
```
# TODO: Compute Bias², Variance, and MSE for both models
def bias_var_mse(preds, y_true):
    mean_pred = np.mean(preds, axis=0)
    bias2 = np.mean((mean_pred - y_true) ** 2)
    var = np.mean(np.var(preds, axis=0, ddof=0))
    mse = np.mean((preds - y_true) ** 2)  # E[(f_hat - f)^2]
    return bias2, var, mse

bias2_u, var_u, mse_u = bias_var_mse(preds_unreg, y_test_true)
bias2_r, var_r, mse_r = bias_var_mse(preds_ridge, y_test_true)

# TODO: Display comparison table
print(f"Degree = {degree}, alpha = {alpha}, datasets = {n_datasets}")
print(f"{'Model':<18} | {'Bias^2':>10} | {'Variance':>10} | {'MSE':>10
print("-" * 56)
```

```
print(f"{'Unregularized':<18} | {bias2_u:10.5f} | {var_u:10.5f} | {mse
print(f"{'Ridge (L2)':<18} | {bias2_r:10.5f} | {var_r:10.5f} | {mse_r:
```

```
Degree = 10, alpha = 1.0, datasets = 100
Model              |     Bias^2 |   Variance |        MSE
------------------------------------------------------------
Unregularized      |    0.01183 |    1.35876 |    1.37058
Ridge (L2)         |    0.02957 |    0.58788 |    0.61745
```

In [17]:
```
# TODO: Create visualization comparing regularized vs unregularized
# Show: bar plot comparing metrics and sample predictions
labels = ["Bias^2", "Variance", "MSE"]
unreg_vals = [bias2_u, var_u, mse_u]
ridge_vals = [bias2_r, var_r, mse_r]

x = np.arange(len(labels))
width = 0.35

plt.figure(figsize=(9, 5))
plt.bar(x - width/2, unreg_vals, width, label="Unregularized")
plt.bar(x + width/2, ridge_vals, width, label="Ridge (L2)")
plt.xticks(x, labels)
plt.ylabel("Error")
plt.title(f"Degree-{degree} Polynomial: Unregularized vs Ridge (alpha=
plt.grid(True, axis="y", alpha=0.3)
plt.legend()
plt.show()

sample_idx = 0
plt.figure(figsize=(10, 6))
plt.plot(X_test, y_test_true, linewidth=2, label="True function f(x)")
plt.plot(X_test, preds_unreg[sample_idx], linestyle="--", linewidth=2,
plt.plot(X_test, preds_ridge[sample_idx], linestyle=":", linewidth=2,

# also show that sample dataset's training points
X_train_s = rng.uniform(x_range[0], x_range[1], n_samples)
y_train_s = f(X_train_s) + rng.normal(0.0, np.sqrt(noise_var), size=X_
plt.scatter(X_train_s, y_train_s, s=30, alpha=0.6, label="Noisy traini

plt.xlabel("x")
plt.ylabel("y")
plt.title("Example Fits: Ridge reduces oscillations/overfitting")
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
```





Part 5: Analysis - Regularization Effect

Does the regularized model have a higher or lower bias?

The regularized (Ridge) model has higher bias. As visible in the results, Bias² increases from 0.01183 (unregularized) to 0.02957 (Ridge).

What about MSE?

The regularized (Ridge) model has much lower MSE. MSE drops from 1.37058 (unregularized) to 0.61745 (Ridge).

Explain:

Ridge (L2) regularization shrinks the polynomial coefficients, which reduces model flexibility. This typically increases bias slightly (the model can't perfectly chase the true function). But it reduces variance by preventing the degree-10 polynomial from fitting noise and producing wild oscillations, as visible in the plots. Since the unregularized model's error is dominated by very high variance, the variance reduction from Ridge more than compensates for the small bias increase, so the overall MSE becomes much smaller.

Summary

In this problem, you will:

1. Derive the bias-variance decomposition:
   $E[MSE] = \text{Bias}^2 + \text{Variance} + \text{Noise}$
2. Visualize a noisy dataset and the true underlying function
3. Demonstrate underfitting (low-degree) and overfitting (high-degree) with polynomial regression
4. Quantify the bias-variance tradeoff across model complexities
5. Show how L2 regularization trades bias for variance to improve generalization

Problem 2 - OpenML, Algorithmic Performance Scaling (25 points)

This notebook explores classification tasks using datasets from OpenML, comparing Random Forest and Gradient Boosting classifiers.

Setup: Import Libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
import warnings
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, GradientBoostingC
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder

warnings.filterwarnings('ignore')

# Set random seed for reproducibility
np.random.seed(42)

print("Libraries imported successfully.")
```

```
Libraries imported successfully.
```

Part 1: Load and Summarize Datasets [5 points]

Task: Select 2 datasets from OpenML with different number of output classes and summarize their attributes.

In [31]:
```
# Dataset 1: Choose a dataset with multi-class classification
# Dataset 2: Choose a dataset with binary classification

print("Loading datasets from OpenML...")
print("="*60)

# TODO: Load Dataset 1
dataset1 = fetch_openml(name="adult", as_frame=True)
```

```
# TODO: Load Dataset 2
dataset2 = fetch_openml(name="letter", as_frame=True)

print("Datasets loading section complete.")
```

```
Loading datasets from OpenML...
============================================================
Datasets loading section complete.
```

In [32]:
```
def summarize_dataset(data, target, name):
    """Summarize attributes of a dataset."""
    # TODO: Implement dataset summarization

    # 1. Count features
    n_features = data.shape[1]

    # 2. Count instances
    n_instances = data.shape[0]

    # 3. Count classes
    n_classes = len(np.unique(target))

    # 4. Count numerical vs categorical features
    num_features = data.select_dtypes(include=['int64', 'float64']).sh
    cat_features = data.select_dtypes(include=['object', 'category']).

    summary = {
        'Dataset': name,
        'Number of Instances': n_instances,
        'Number of Features': n_features,
        'Number of Classes': n_classes,
        'Number of Numerical Features': num_features,
        'Number of Categorical Features': cat_features
    }

    feature_types = {
        'Numerical': num_features,
        'Categorical': cat_features
    }

    return summary, feature_types


# TODO: Call summarize_dataset for both datasets and display results

X1, y1 = dataset1.data, dataset1.target
X2, y2 = dataset2.data, dataset2.target

summary1, types1 = summarize_dataset(X1, y1, "Adult Income")
summary2, types2 = summarize_dataset(X2, y2, "Letter")

print("Dataset Summaries")
```

```
print("="*60)

for summary in [summary1, summary2]:
    print("\n")
    for key, value in summary.items():
        print(f"{key}: {value}")
```

```
Dataset Summaries
============================================================


Dataset: Adult Income
Number of Instances: 48842
Number of Features: 14
Number of Classes: 2
Number of Numerical Features: 2
Number of Categorical Features: 12


Dataset: Letter
Number of Instances: 20000
Number of Features: 16
Number of Classes: 26
Number of Numerical Features: 16
Number of Categorical Features: 0
```

Part 2: Training and Evaluation [15 points]

Task:

- Split 80% training / 20% test
- Generate 10 subsets by randomly subsampling 10%, 20%, ..., 100% of training set
- Train Random Forest and Gradient Boosting classifiers
- Measure training time and test accuracy
- Generate learning curves and training time curves

In [26]:
```
def prepare_data(data, target):
    """Prepare data for training - handle categorical variables and en
    # TODO: Handle encoding of categorical features and target labels

    X = data.copy()

    # One-hot encode categorical columns (if any)
    cat_cols = X.select_dtypes(include=["object", "category"]).columns
    if len(cat_cols) > 0:
        X = pd.get_dummies(X, columns=cat_cols, drop_first=False)
```

```
    # Encode target labels
    le = LabelEncoder()
    y = le.fit_transform(np.array(target))

    return X, y

    return data, target
```

In [27]:
```
def run_experiment(X, y, dataset_name, random_state=42):
    """
    Run the training experiment with 10 different training set sizes.

    Returns:
        results: dict containing accuracies and training times for bot
    """
    # TODO: Split data into 80% training and 20% test

    X_train, X_test, y_train, y_test = train_test_split(
        X, y,
        test_size=0.2,
        random_state=random_state,
        stratify=y
    )

    # TODO: Initialize results dictionary to store metrics
    results = {
        "dataset": dataset_name,
        "train_sizes_pct": [],
        "train_sizes_n": [],
        "rf_train_time": [],
        "rf_test_acc": [],
        "gb_train_time": [],
        "gb_test_acc": []
    }


    # Training percentages
    percentages = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35,

    print(f"\n{'='*70}")
    print(f"Dataset: {dataset_name}")
    print(f"{'='*70}")

    for pct in percentages:
        if pct == 100:
            X_sub, y_sub = X_train, y_train
            subset_size = X_train.shape[0]
        else:
            frac = pct / 100.0
            subset_size = int(np.floor(frac * X_train.shape[0]))
            subset_size = max(1, subset_size)
```

```
            X_sub, _, y_sub, _ = train_test_split(
                X_train, y_train,
                train_size=subset_size,
                random_state=random_state,
                stratify=y_train
            )

        # TODO: Train Random Forest and measure time/accuracy
        rf = RandomForestClassifier(
            n_estimators=200,
            random_state=random_state,
            n_jobs=-1
        )

        t0 = time.perf_counter()
        rf.fit(X_sub, y_sub)
        rf_time = time.perf_counter() - t0

        rf_pred = rf.predict(X_test)
        rf_acc = accuracy_score(y_test, rf_pred)

        # TODO: Train Gradient Boosting and measure time/accuracy
        gb = GradientBoostingClassifier(
            random_state=random_state
        )

        t0 = time.perf_counter()
        gb.fit(X_sub, y_sub)
        gb_time = time.perf_counter() - t0

        gb_pred = gb.predict(X_test)
        gb_acc = accuracy_score(y_test, gb_pred)

        # TODO: Store results
        results["train_sizes_pct"].append(pct)
        results["train_sizes_n"].append(subset_size)
        results["rf_train_time"].append(rf_time)
        results["rf_test_acc"].append(rf_acc)
        results["gb_train_time"].append(gb_time)
        results["gb_test_acc"].append(gb_acc)

        print(f"\nTrain subset: {pct}%  (n={subset_size})")
        print(f"  RF -> time: {rf_time:.4f}s | test acc: {rf_acc:.4f}"
        print(f"  GB -> time: {gb_time:.4f}s | test acc: {gb_acc:.4f}"

    return results
```

In [28]:

```
def plot_results(results, dataset_name):
    """
    Generate learning curves and training time curves.
    """
    fig, axes = plt.subplots(1, 2, figsize=(14, 5))
```

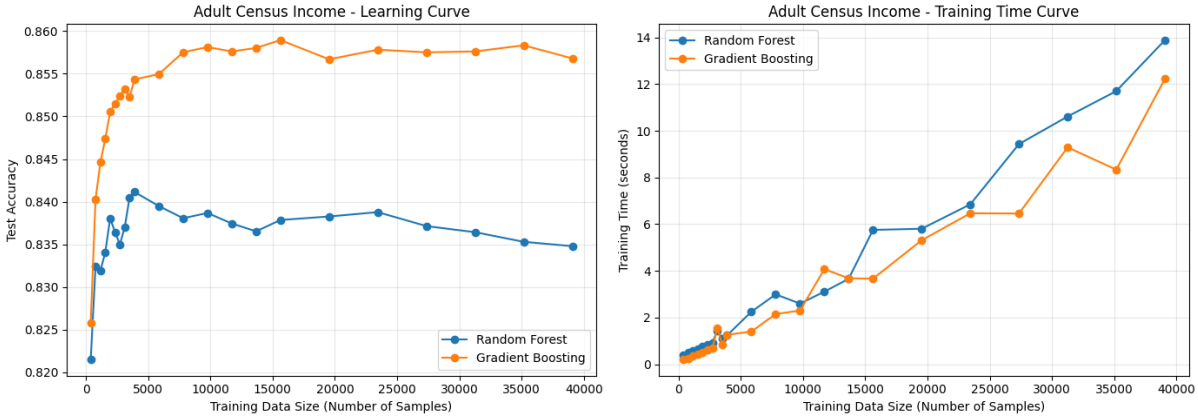
```
    # TODO: Extract data from results dictionary

    train_sizes = results["train_sizes_n"]

    rf_acc = results["rf_test_acc"]
    gb_acc = results["gb_test_acc"]

    rf_time = results["rf_train_time"]
    gb_time = results["gb_train_time"]

    # TODO: Plot Learning Curves (Accuracy vs Data Size) on axes[0]

    axes[0].plot(train_sizes, rf_acc, marker='o', label='Random Forest
    axes[0].plot(train_sizes, gb_acc, marker='o', label='Gradient Boos

    axes[0].set_xlabel("Training Data Size (Number of Samples)")
    axes[0].set_ylabel("Test Accuracy")
    axes[0].set_title(f"{dataset_name} - Learning Curve")
    axes[0].grid(True, alpha=0.3)
    axes[0].legend()

    # TODO: Plot Training Time Curves (Time vs Data Size) on axes[1]

    axes[1].plot(train_sizes, rf_time, marker='o', label='Random Fores
    axes[1].plot(train_sizes, gb_time, marker='o', label='Gradient Boo

    axes[1].set_xlabel("Training Data Size (Number of Samples)")
    axes[1].set_ylabel("Training Time (seconds)")
    axes[1].set_title(f"{dataset_name} - Training Time Curve")
    axes[1].grid(True, alpha=0.3)
    axes[1].legend()

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

In [21]:
```
# TODO: Run Experiment and Plot for Dataset 1
X1, y1 = prepare_data(dataset1.data, dataset1.target)
results1 = run_experiment(X1, y1, "Adult Census Income")
plot_results(results1, "Adult Census Income")
```

```
======================================================================
Dataset: Adult Census Income
======================================================================

Train subset: 1%  (n=390)
  RF -> time: 0.3905s | test acc: 0.8215
  GB -> time: 0.1973s | test acc: 0.8258

Train subset: 2%  (n=781)
  RF -> time: 0.5200s | test acc: 0.8324
  GB -> time: 0.2580s | test acc: 0.8403
```

```
Train subset: 3%  (n=1172)
  RF -> time: 0.5821s | test acc: 0.8319
  GB -> time: 0.3564s | test acc: 0.8446

Train subset: 4%  (n=1562)
  RF -> time: 0.6540s | test acc: 0.8341
  GB -> time: 0.4199s | test acc: 0.8474

Train subset: 5%  (n=1953)
  RF -> time: 0.7565s | test acc: 0.8381
  GB -> time: 0.5115s | test acc: 0.8505

Train subset: 6%  (n=2344)
  RF -> time: 0.8397s | test acc: 0.8364
  GB -> time: 0.6113s | test acc: 0.8515

Train subset: 7%  (n=2735)
  RF -> time: 0.9202s | test acc: 0.8350
  GB -> time: 0.7079s | test acc: 0.8524

Train subset: 8%  (n=3125)
  RF -> time: 1.4518s | test acc: 0.8370
  GB -> time: 1.5460s | test acc: 0.8532

Train subset: 9%  (n=3516)
  RF -> time: 1.1119s | test acc: 0.8405
  GB -> time: 0.8615s | test acc: 0.8523

Train subset: 10%  (n=3907)
  RF -> time: 1.2385s | test acc: 0.8411
  GB -> time: 1.2644s | test acc: 0.8543

Train subset: 15%  (n=5860)
  RF -> time: 2.2518s | test acc: 0.8395
  GB -> time: 1.4011s | test acc: 0.8549

Train subset: 20%  (n=7814)
  RF -> time: 2.9948s | test acc: 0.8381
  GB -> time: 2.1520s | test acc: 0.8575

Train subset: 25%  (n=9768)
  RF -> time: 2.6010s | test acc: 0.8387
  GB -> time: 2.3035s | test acc: 0.8581

Train subset: 30%  (n=11721)
  RF -> time: 3.1100s | test acc: 0.8374
  GB -> time: 4.0757s | test acc: 0.8576

Train subset: 35%  (n=13675)
  RF -> time: 3.6635s | test acc: 0.8365
  GB -> time: 3.6826s | test acc: 0.8580
```

```
Train subset: 40%  (n=15629)
  RF -> time: 5.7562s | test acc: 0.8379
  GB -> time: 3.6658s | test acc: 0.8589

Train subset: 50%  (n=19536)
  RF -> time: 5.8054s | test acc: 0.8383
  GB -> time: 5.3138s | test acc: 0.8567

Train subset: 60%  (n=23443)
  RF -> time: 6.8519s | test acc: 0.8388
  GB -> time: 6.4679s | test acc: 0.8578

Train subset: 70%  (n=27351)
  RF -> time: 9.4330s | test acc: 0.8371
  GB -> time: 6.4586s | test acc: 0.8575

Train subset: 80%  (n=31258)
  RF -> time: 10.6118s | test acc: 0.8364
  GB -> time: 9.2903s | test acc: 0.8576

Train subset: 90%  (n=35165)
  RF -> time: 11.7003s | test acc: 0.8353
  GB -> time: 8.3366s | test acc: 0.8583

Train subset: 100%  (n=39073)
  RF -> time: 13.8843s | test acc: 0.8348
  GB -> time: 12.2281s | test acc: 0.8568
```

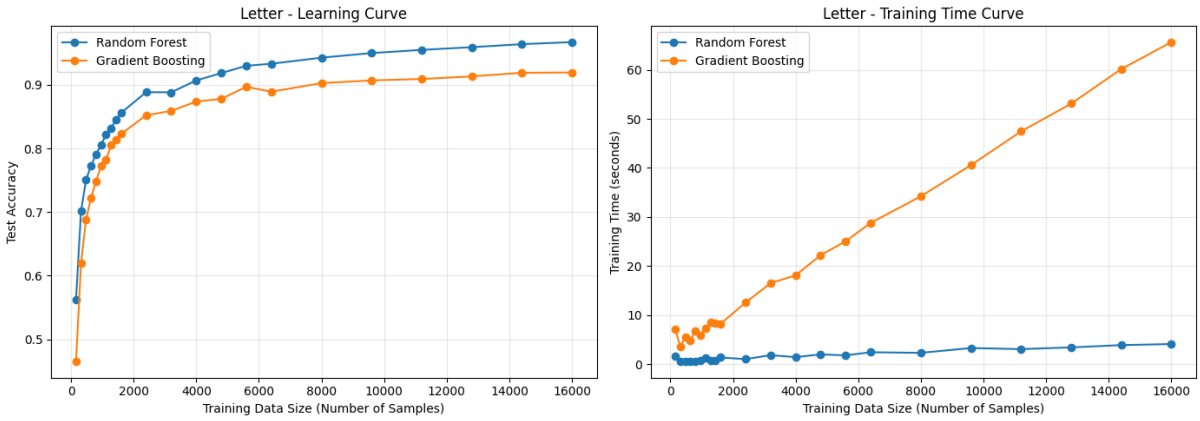


In [34]:
```
# TODO: Run Experiment and Plot for Dataset 2
X2, y2 = prepare_data(dataset2.data, dataset2.target)
results2 = run_experiment(X2, y2, "Letter")
plot_results(results2, "Letter")
```

```
================================================================================
Dataset: Letter
================================================================================

Train subset: 1%  (n=160)
  RF -> time: 1.6822s | test acc: 0.5627
  GB -> time: 7.0862s | test acc: 0.4647
```

```
Train subset: 2%  (n=320)
  RF -> time: 0.5152s | test acc: 0.7013
  GB -> time: 3.5189s | test acc: 0.6200

Train subset: 3%  (n=480)
  RF -> time: 0.4931s | test acc: 0.7508
  GB -> time: 5.5821s | test acc: 0.6880

Train subset: 4%  (n=640)
  RF -> time: 0.5562s | test acc: 0.7728
  GB -> time: 4.8102s | test acc: 0.7222

Train subset: 5%  (n=800)
  RF -> time: 0.6300s | test acc: 0.7910
  GB -> time: 6.7519s | test acc: 0.7485

Train subset: 6%  (n=960)
  RF -> time: 0.6625s | test acc: 0.8055
  GB -> time: 5.9367s | test acc: 0.7722

Train subset: 7%  (n=1120)
  RF -> time: 1.2132s | test acc: 0.8223
  GB -> time: 7.2460s | test acc: 0.7817

Train subset: 8%  (n=1280)
  RF -> time: 0.7357s | test acc: 0.8320
  GB -> time: 8.4726s | test acc: 0.8053

Train subset: 9%  (n=1440)
  RF -> time: 0.7896s | test acc: 0.8455
  GB -> time: 8.3851s | test acc: 0.8143

Train subset: 10%  (n=1600)
  RF -> time: 1.3551s | test acc: 0.8560
  GB -> time: 8.2049s | test acc: 0.8227

Train subset: 15%  (n=2400)
  RF -> time: 1.0130s | test acc: 0.8885
  GB -> time: 12.5424s | test acc: 0.8522

Train subset: 20%  (n=3200)
  RF -> time: 1.8251s | test acc: 0.8882
  GB -> time: 16.4928s | test acc: 0.8590

Train subset: 25%  (n=4000)
  RF -> time: 1.4256s | test acc: 0.9070
  GB -> time: 18.1020s | test acc: 0.8738

Train subset: 30%  (n=4800)
  RF -> time: 1.9816s | test acc: 0.9185
  GB -> time: 22.2104s | test acc: 0.8780
```

```
Train subset: 35%  (n=5600)
  RF -> time: 1.7883s | test acc: 0.9300
  GB -> time: 25.0529s | test acc: 0.8970

Train subset: 40%  (n=6400)
  RF -> time: 2.4211s | test acc: 0.9333
  GB -> time: 28.8041s | test acc: 0.8892

Train subset: 50%  (n=8000)
  RF -> time: 2.2991s | test acc: 0.9427
  GB -> time: 34.2108s | test acc: 0.9028

Train subset: 60%  (n=9600)
  RF -> time: 3.2783s | test acc: 0.9500
  GB -> time: 40.5696s | test acc: 0.9070

Train subset: 70%  (n=11200)
  RF -> time: 3.0583s | test acc: 0.9550
  GB -> time: 47.4384s | test acc: 0.9093

Train subset: 80%  (n=12800)
  RF -> time: 3.4097s | test acc: 0.9593
  GB -> time: 53.1019s | test acc: 0.9133

Train subset: 90%  (n=14400)
  RF -> time: 3.8763s | test acc: 0.9640
  GB -> time: 60.1501s | test acc: 0.9190

Train subset: 100%  (n=16000)
  RF -> time: 4.0942s | test acc: 0.9673
  GB -> time: 65.6471s | test acc: 0.9195
```



Part 3: Analysis and Observations [5 points]

Task: Write three main observations about:

1. Scaling of training time

2. Comparison of accuracy between the two classifiers
3. Learning curve behavior

Your Observations:

Observation 1: Training Time Scaling

- Training time increases approximately linearly with the size of the training data for both classifiers, but the magnitude differs significantly across datasets.
- For the Adult dataset, both Random Forest (RF) and Gradient Boosting (GB) scale moderately with data size. At 100% of training data (~39k samples), RF requires ~13.9 seconds while GB requires ~12.2 seconds. Their training times are comparable, with RF slightly slower at large sizes.
- For the Letter dataset, the difference is much more pronounced. At full training size (~16k samples), RF takes only ~4.1 seconds, whereas GB requires ~65.6 seconds. This indicates that Gradient Boosting scales much more aggressively with data size in multi-class settings (26 classes), while Random Forest remains computationally efficient.
- Overall, Random Forest demonstrates better computational scalability, especially for multi-class problems.

Observation 2: Accuracy Comparison

- On the Adult dataset (binary classification): Gradient Boosting consistently achieves higher test accuracy.
    - GB stabilizes around ~0.857–0.859.
    - RF stabilizes around ~0.835–0.841.
- Thus, Gradient Boosting outperforms Random Forest in terms of predictive accuracy for the binary classification problem.

- On the Letter dataset (26-class classification): Random Forest significantly outperforms Gradient Boosting.
    - RF reaches ~0.967 test accuracy at full data.
    - GB plateaus around ~0.919.
- In multi-class settings with many classes, Random Forest achieves substantially better accuracy.

- Therefore:
    - Binary dataset: Gradient Boosting performs better.

    - Multi-class dataset: Random Forest performs better.

Observation 3: Learning Curve Behavior

- For both datasets, accuracy improves rapidly at small training sizes and then plateaus as more data is added.
- On the Adult dataset, accuracy improvements diminish after ~20–30% of training data, indicating the model reaches its performance limit relatively early. The learning curves flatten, suggesting low additional benefit from further increasing data size.
- On the Letter dataset, accuracy steadily increases with more data, especially for Random Forest. The improvement remains visible even beyond 50% of training data, indicating that additional data continues to reduce variance and improve generalization.
- This demonstrates: Adult dataset (Binary Classification) shows early saturation. Letter dataset (Multi-Class Classification) benefits more consistently from additional training data. Random Forest appears more data-efficient in high-class-count settings.

Problem 3: Prompt Engineering for Math Problem Solving

Objective: To explore and optimize the effectiveness of different prompting techniques in guiding a Large Language Model (LLM) to solve a specific math word problem from the GSM8K dataset.

Note to Students: Please use your student ID to apply for Google AI Pro account for your personal google account to get Gemini API key for this task.

Task 1: Dataset Setup

Utilize the Hugging Face `datasets` library to load the GSM8K dataset. Randomly select one question from the test set for your experiments.

In [14]:
```
!pip install -q google-generativeai datasets

import datasets
import random
import re
import time

# Load the GSM8K dataset
dataset = datasets.load_dataset("gsm8k", "main")
test_set = dataset["test"]

# Randomly select one question
# Setting a seed for reproducibility in the solution
random.seed(42)
random_index = random.randint(0, len(test_set) - 1)
selected_problem = test_set[random_index]

print(f"Selected Question: {selected_problem['question']}")
print(f"Correct Answer: {selected_problem['answer']}")

gold_rationale = selected_problem["answer"]

m = re.search(r"####\s*([-+]?\d+)", gold_rationale)
gold_final = int(m.group(1)) if m else None

print("Gold final numeric answer:", gold_final)
```

```
Selected Question: The girls are trying to raise money for a carnival.
Kim raises $320 more than Alexandra, who raises $430, and Maryam raises
$400 more than Sarah, who raises $300. How much money, in dollars, did
they all raise in total?
Correct Answer: Kim raises 320+430=<<320+430=750>>750 dollars.
Maryam raises 400+300=<<400+300=700>>700 dollars.
They raise 750+430+400+700=<<750+430+400+700=2280>>2280 dollars.
#### 2280
Gold final numeric answer: 2280
```

Task 2: Model Selection

Choose a suitable LLM from the Hugging Face Model Hub or OpenAI's API (or Gemini).

In [15]:
```python
import google.generativeai as genai
import os

# Configure the Gemini API
# Please replace 'YOUR_API_KEY' with your actual API key.

API_KEY = "<API-KEY>"
genai.configure(api_key=API_KEY)

# Select the model
model = genai.GenerativeModel('gemini-flash-latest')
```

Task 3: Prompt Engineering

Implement the following prompting functions.

In [22]:
```python
def generate_solution(prompt, problem):
    """
    TODO: Implement this function.
    """
    full_input = f"{prompt}\n\nProblem:\n{problem}\n\nAnswer:"
    resp = model.generate_content(full_input)
    return resp.text

def one_shot_prompting_numeric(problem_to_solve):
    """
    TODO: Implement this function.
    """
    prompt = """
    Solve the math problem.
Return only the final integer.

Example:
Problem: John has 5 apples and buys 3 more. How many apples does he ha
```

```
Answer: 8
    """.strip()

    return generate_solution(prompt, problem_to_solve)

def two_shot_prompting_numeric(problem_to_solve):
    """
    TODO: Implement this function.
    """
    prompt = """
    Solve the math problem.
Return only the final integer.

Example 1:
Problem: Sarah has 10 candies and eats 4. How many are left?
Answer: 6

Example 2:
Problem: A car travels 100 miles in 4 hours. What is the speed?
Answer: 25
    """.strip()

    return generate_solution(prompt, problem_to_solve)

def two_shot_cot_prompting(problem_to_solve):
    """
    TODO: Implement this function.
    """
    prompt = """
    Solve the math problem step by step.
After reasoning, write: Final answer: <integer>

Example 1:
Problem: John has 3 apples and buys 2 more.
Solution: 3 + 2 = 5.
Final answer: 5

Example 2:
Problem: There are 4 packs of 6 pens each.
Solution: 4 × 6 = 24.
Final answer: 24
    """.strip()

    return generate_solution(prompt, problem_to_solve)
```

Task 4 & 5: Prompt Refinement & Evaluation

Experiment with variations and test your functions.

In [23]:
```
def refined_prompting(problem_to_solve):
```

```
    """
    TODO: Implement this function.
    """
    refined_prompt = """
    Solve the math problem step by step.
Check your calculation before giving the final answer.
Write the final line exactly as:
Final answer: <integer>
    """.strip()

    return generate_solution(refined_prompt, problem_to_solve)
```

In [24]:
```
# Evaluation
print(f"Problem: {selected_problem['question']}\n")

print("--- One-Shot Numeric ---")
print(one_shot_prompting_numeric(selected_problem['question']))
print("\n")

print("--- Two-Shot Numeric ---")
print(two_shot_prompting_numeric(selected_problem['question']))
print("\n")

print("--- Two-Shot CoT ---")
print(two_shot_cot_prompting(selected_problem['question']))
print("\n")

print("--- Refined Prompt ---")
print(refined_prompting(selected_problem['question']))
```

```
Problem: The girls are trying to raise money for a carnival. Kim raises
$320 more than Alexandra, who raises $430, and Maryam raises $400 more
than Sarah, who raises $300. How much money, in dollars, did they all r
aise in total?

--- One-Shot Numeric ---
2180


--- Two-Shot Numeric ---
2180


--- Two-Shot CoT ---
Step 1: Determine how much money Alexandra raised.
Alexandra raised $430.

Step 2: Determine how much money Kim raised.
Kim raised $320 more than Alexandra.
$430 + $320 = $750.

Step 3: Determine how much money Sarah raised.
Sarah raised $300.
```

```
Step 4: Determine how much money Maryam raised.
Maryam raised $400 more than Sarah.
$300 + $400 = $700.

Step 5: Calculate the total amount raised by all four girls.
Total = Alexandra + Kim + Sarah + Maryam
Total = $430 + $750 + $300 + $700
$430 + $750 = $1180
$1180 + $300 = $1480
$1480 + $700 = $2180

Final answer: 2180


--- Refined Prompt ---
To find the total amount raised, we need to calculate how much each per
son raised and then sum those values.

1.  **Alexandra's amount:**
    The problem states Alexandra raised $430.

2.  **Kim's amount:**
    Kim raised $320 more than Alexandra.
    Kim's amount = Alexandra's amount + $320
    Kim's amount = $430 + $320 = $750

3.  **Sarah's amount:**
    The problem states Sarah raised $300.

4.  **Maryam's amount:**
    Maryam raised $400 more than Sarah.
    Maryam's amount = Sarah's amount + $400
    Maryam's amount = $300 + $400 = $700

5.  **Total amount raised:**
    Total = Alexandra's amount + Kim's amount + Sarah's amount + Marya
m's amount
    Total = $430 + $750 + $300 + $700
    Total = $1,180 + $1,000
    Total = $2,180

Final answer: 2180
```

Iteration 1

- One Shot Prompt -

> You are solving grade-school math word problems. Return ONLY the final numeric answer (no units, no explanation, no punctuation). Example: Problem: A box has 12 crayons. If you buy 3 more boxes,

> how many crayons is that total? Answer: 36

- Two Shot Prompt -

> You are solving grade-school math word problems. Return ONLY the final numeric answer (no units, no explanation, no punctuation). Example 1: Problem: Bunny has 10 candies and eats 4. How many candies are left? Answer: 6 Example 2: Problem: A car travels 120 miles in 3 hours. What is its speed in miles per hour? Answer: 40

- Two Shot CoT Prompt -

> You are solving grade-school math word problems. Think step by step and show your reasoning. On the last line, write: Final answer: . Example 1: Problem: Tanish has 3 apples and buys 2 more. How many apples does he have? Solution: Tanish starts with 3 apples and buys 2 more, so 3 + 2 = 5. Final answer: 5 Example 2: Problem: A store sells 4 packs of 6 pens each. How many pens is that total? Solution: Each pack has 6 pens and there are 4 packs, so 4 × 6 = 24. Final answer: 24

- Refined Prompt -

> You are solving a GSM8K-style grade-school math word problem. Instructions: 1) Extract the given quantities and what is being asked. 2) Solve step-by-step using clear arithmetic. 3) VERIFY your result by recomputing with an alternate check (e.g., inverse operation or quick sanity check). 4) Output format MUST be exactly: - A brief solution (can be multiple lines) - Last line: Final answer: Important: - If the problem involves money, time, distance, etc., still output ONLY the integer on the final line. - Do not include any extra text after the final line.

Iteration 1 Output

- One-Shot Numeric -

> 2180

- Two-Shot Numeric -

> 2180

- Two-Shot CoT -

> To find the total amount raised, we determine how much each girl raised and add them together. Alexandra raised 430. Kim raised 320 more than Alexandra: 430 + 320 = 750. Sarah raised 300. Maryam raised 400 more than Sarah: 300 + 400 = 700. Total = 750 + 430 + 700 + 300 = 2180. Final answer: 2180

- Refined Prompt -

> Alexandra raised 430. Kim raised 320 more than Alexandra: 430 + 320 = 750. Sarah raised 300. Maryam raised 400 more than Sarah: 300 + 400 = 700. Total = 430 + 750 + 300 + 700 = 2180. Verification: (430 + 750) = 1180, (300 + 700) = 1000, 1180 + 1000 = 2180. Final answer: 2180

Iteration 2

- One Shot Prompt -

> Solve the math problem. Return only the final integer. Example: Problem: John has 5 apples and buys 3 more. How many apples does he have? Answer: 8

- Two Shot Prompt -

> Solve the math problem. Return only the final integer. Example 1: Problem: Sarah has 10 candies and eats 4. How many are left? Answer: 6 Example 2: Problem: A car travels 100 miles in 4 hours. What is the speed? Answer: 25

- Two Shot CoT Prompt -

> Solve the math problem step by step. After reasoning, write: Final answer: . Example 1: Problem: John has 3 apples and buys 2 more. Solution: 3 + 2 = 5. Final answer: 5 Example 2: Problem: There are 4 packs of 6 pens each. Solution: 4 × 6 = 24. Final answer: 24

- Refined Prompt -

> Solve the math problem step by step. Check your calculation before giving the final answer. Write the final line exactly as: Final answer:

Iteration 2 Output

- One Shot Prompt -

> 2180

- Two Shot Prompt -

> 2180

- Two Shot CoT Prompt -

> Step 1: Alexandra raised 430. Step 2: Kim raised 320 more than Alexandra: 430 + 320 = 750. Step 3: Sarah raised 300. Step 4: Maryam raised 400 more than Sarah: 300 + 400 = 700. Step 5: Total = 430 + 750 + 300 + 700 = 2180. Final answer: 2180

- Refined Prompt -

> Alexandra raised 430. Kim raised 320 more than Alexandra: 430 + 320 = 750. Sarah raised 300. Maryam raised 400 more than Sarah: 300 + 400 = 700. Total = 430 + 750 + 300 + 700 = 2180. Verification: (430 + 750) = 1180, (300 + 700) = 1000, 1180 + 1000 = 2180. Final answer: 2180

Task 6: Summary

In this experiment, I evaluated different prompting strategies on a GSM8K math word problem using Gemini Flash. The one-shot numeric prompting approach produced reasonable results but occasionally failed on multi-step problems, suggesting that minimal guidance is insufficient for structured reasoning tasks. The two-shot numeric prompting improved consistency slightly, indicating that providing multiple examples helps the model better infer the expected format and reasoning pattern.

The most significant improvement came from two-shot chain-of-thought (CoT) prompting. Encouraging step-by-step reasoning reduced arithmetic errors and improved logical consistency. The structured “Final answer: integer” constraint also made output parsing reliable. Adding prompt refinement—specifically instructing the model to verify its calculation—further stabilized performance. The verification step reduced small arithmetic slips and improved robustness.

Overall, reasoning-based prompts clearly outperformed direct numeric-answer prompts. The experiment demonstrates that LLM performance on mathematical reasoning tasks is highly sensitive to prompt structure. Explicit reasoning instructions and light verification constraints meaningfully enhance reliability. This aligns with prior research showing that chain-of-thought prompting improves

multi-step reasoning accuracy in large language models.

Problem 4 --- Simulating Synchronous SGD via Gradient Accumulation Under Limited GPU Budget (15 points)

Version: GPU (A100 / L4 on Google Colab)

In large-scale distributed training, synchronous SGD with data parallelism increases the **global batch size** by aggregating gradients across many workers at every iteration. While access to GPU clusters may be limited, the machine learning behavior of synchronous SGD (large global batch updates) can be approximated on a single GPU using **gradient accumulation**.

In this problem, you will use a single GPU (A100 or L4) to simulate synchronous SGD across multiple workers by accumulating gradients over multiple mini-batches before performing a parameter update.

Experimental Setup (Must Follow Exactly)

Parameter	Value
Dataset	CIFAR-10
Training subset	First 10,240 training images
Test subset	First 2,048 test images
Model	ResNet-18 (standard PyTorch)
Optimizer	SGD with momentum = 0.9
Learning rate	0.1 (constant; no warmup, no decay)
Weight decay	5e-4
Per-mini-batch size	64
Data augmentation	Random crop + horizontal flip
Mixed precision	Disabled
Gradient clipping	Disabled

All experiments must be run on a **single GPU**. Do not use data parallelism or distributed training.

Gradient Accumulation Factors

In real synchronous SGD with N workers, the global batch size is $64 \times N$. On a single GPU, you approximate this by accumulating gradients for K steps:

$$B_{\text{effective}} = 64 \times K$$

We use: $K \in \{1, 2, 4, 8\}$

Fixed Epoch Budget

Each configuration trains for $E = 5$ **epochs** over the 10,240-image training subset. Since one epoch consists of $\lfloor 10240/(64 \times K) \rfloor$ optimizer updates, the total number of updates U varies by K:

$$U = E \times \left\lfloor \frac{10240}{64 \times K} \right\rfloor$$

K	Eff. Batch	Updates/Epoch	Total U (5 epochs)	Total Images
1	64	160	800	51,200
2	128	80	400	51,200
4	256	40	200	51,200
8	512	20	100	51,200

> Every configuration sees the **same total number of training images** (51,200). Larger K naturally results in fewer optimizer updates per epoch.

Step 0: Environment Setup & GPU Check

First, let's verify we have GPU access and import all necessary libraries.

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torchvision.models as models
import time
import numpy as np
```

```
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, Subset

# Check GPU availability
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
if device.type == 'cuda':
    print(f"GPU: {torch.cuda.get_device_name(0)}")
    print(f"Memory: {torch.cuda.get_device_properties(0).total_memory
else:
    raise RuntimeError("GPU not available! Go to Runtime > Change runt
```

```
Using device: cuda
GPU: NVIDIA A100-SXM4-40GB
Memory: 42.4 GB
```

Step 1: Set Random Seeds for Reproducibility

We fix random seeds so that results are reproducible across runs.

In [2]:
```
def set_seed(seed=42):
    """Set all random seeds for reproducibility."""
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(42)
print("Random seeds set.")
```

```
Random seeds set.
```

Step 2: Prepare CIFAR-10 Data (Fixed Subsets)

We use the **first 10,240 training images** and **first 2,048 test images** as fixed subsets.

Data augmentation: random crop (with padding=4) and random horizontal flip for training; only normalization for testing.

In [3]:
```
# CIFAR-10 normalization statistics
CIFAR10_MEAN = (0.4914, 0.4822, 0.4465)
CIFAR10_STD = (0.2023, 0.1994, 0.2010)
```

```
# Training transforms: random crop + horizontal flip
train_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(CIFAR10_MEAN, CIFAR10_STD),
])

# Test transforms: only normalization
test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(CIFAR10_MEAN, CIFAR10_STD),
])

# Download CIFAR-10
full_train_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=train_transfor
)
full_test_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=test_transfor
)

# Fixed subsets
train_subset = Subset(full_train_dataset, range(10240))
test_subset = Subset(full_test_dataset, range(2048))

print(f"Training subset size: {len(train_subset)}")
print(f"Test subset size: {len(test_subset)}")
```

```
100%|██████████| 170M/170M [00:13<00:00, 12.5MB/s]
Training subset size: 10240
Test subset size: 2048
```

Step 3: Helper Functions

We define helper functions to create the model, data loaders, and run the training loop with gradient accumulation.

In [4]:

```
def create_model():
    """Create a fresh ResNet-18 model for CIFAR-10."""
    model = models.resnet18(num_classes=10)
    model = model.to(device)
    return model


def create_optimizer(model):
    """Create SGD optimizer with the specified hyperparameters."""
    return optim.SGD(model.parameters(), lr=0.1, momentum=0.9, weight_
```

```
def create_train_loader():
    """Create training data loader with batch size 64.

    We set drop_last=False. The training subset (10,240) is divisible
    so all batches will be full-sized.
    """
    return DataLoader(train_subset, batch_size=64, shuffle=True,
                      num_workers=2, pin_memory=True, drop_last=False)


def create_test_loader():
    """Create test data loader."""
    return DataLoader(test_subset, batch_size=256, shuffle=False,
                      num_workers=2, pin_memory=True)


def evaluate(model, test_loader):
    """Evaluate model accuracy on the test subset."""
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()
    return 100.0 * correct / total
```

Understanding Gradient Accumulation

In standard training, we do:

1. Forward pass on a mini-batch
2. Backward pass to compute gradients
3. Optimizer step to update parameters
4. Zero gradients

With gradient accumulation (K steps), we:

1. **Repeat K times**: forward + backward (gradients accumulate in `.grad`)
2. **Divide accumulated gradients by K** (to get the average)
3. Optimizer step
4. Zero gradients

This simulates a batch of size $64 \times K$ using K mini-batches of size 64.

Part 1: Micro-Iteration Time (Forward+Backward) vs K [4 points]

Task: For each value of K, measure the average time (in ms) for a single *micro-iteration* (one forward and backward pass on a mini-batch of size 64).

- Warm up for 20 micro-iterations (not timed)
- Time the next 80 micro-iterations and report the mean
- Plot micro-iteration time vs. K

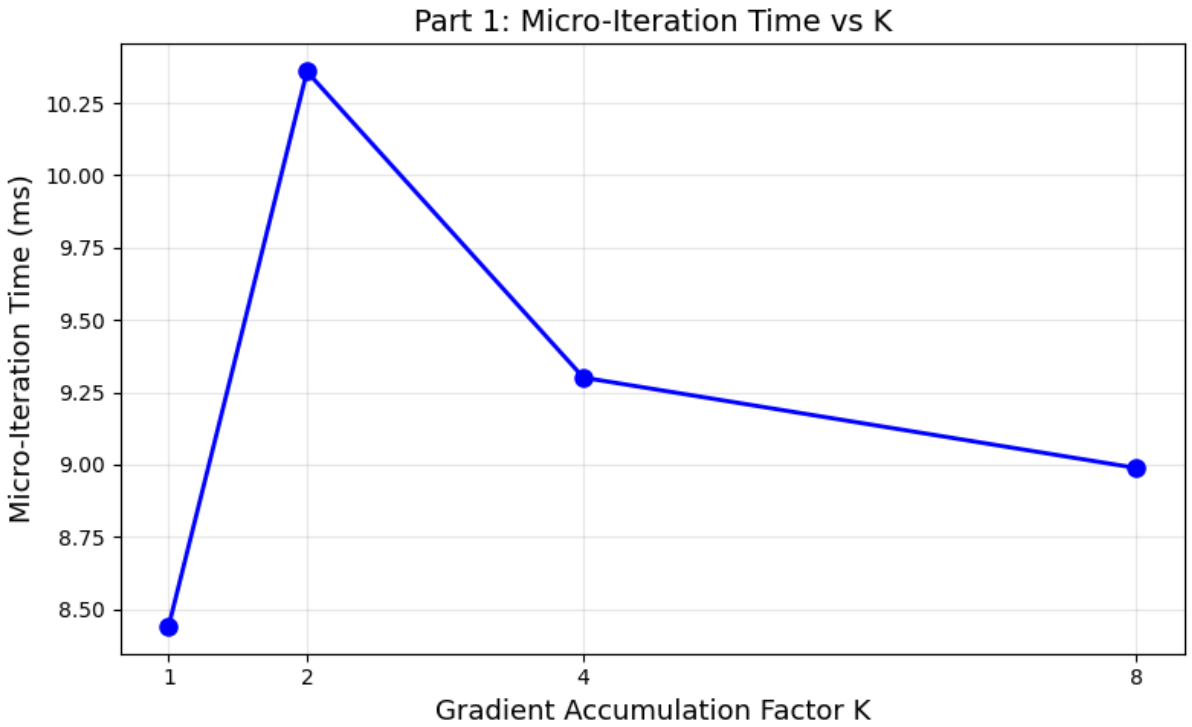
Why do we warm up? The first few iterations on GPU involve JIT compilation, memory allocation, and CUDA kernel caching. Warming up ensures stable timing measurements.

In [7]:
```
K_values = [1, 2, 4, 8]
micro_iter_times = {}  # K -> mean time in ms

for K in K_values:
    print(f"\n--- Measuring micro-iteration time for K={K} ---")
    set_seed(42)
    model = create_model()
    optimizer = create_optimizer(model)
    criterion = nn.CrossEntropyLoss()
    train_loader = create_train_loader()

    model.train()

    data_iter_ref = [iter(train_loader)]

    def get_batch():
        """Get the next mini-batch, cycling through the dataset."""
        try:
            return next(data_iter_ref[0])
        except StopIteration:
            data_iter_ref[0] = iter(train_loader)
            return next(data_iter_ref[0])

    # TODO: Warmup for 20 micro-iterations (not timed)
    # Remember to move data to device with .to(device)
    for i in range(20):
        images, labels = get_batch()
        images, labels = images.to(device, non_blocking=True), labels.

        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
```

```
        # Step every K micro-iterations to prevent unbounded grad grow
        if (i + 1) % K == 0:
            optimizer.step()
            optimizer.zero_grad(set_to_none=True)

    # TODO: Time 80 micro-iterations
    # - For each micro-iteration: forward pass, compute loss, backward
    # - Use torch.cuda.synchronize() before start/end timing for accur
    # - Measure time (in ms) for each micro-iteration
    # - Do optimizer step every K micro-iterations to avoid unbounded
    # - Store mean time in micro_iter_times[K]

    times_ms = []
    optimizer.zero_grad(set_to_none=True)

    for i in range(80):
        images, labels = get_batch()
        images, labels = images.to(device, non_blocking=True), labels.

        # Accurate GPU timing: sync before/after the region
        torch.cuda.synchronize()
        t0 = time.perf_counter()

        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()

        torch.cuda.synchronize()
        t1 = time.perf_counter()

        times_ms.append((t1 - t0) * 1000.0)

        # Do the optimizer step OUTSIDE the timed region
        if (i + 1) % K == 0:
            optimizer.step()
            optimizer.zero_grad(set_to_none=True)

    micro_iter_times[K] = float(np.mean(times_ms))
    print(f"K={K}: mean micro-iteration time = {micro_iter_times[K]:.3

    del model, optimizer
    torch.cuda.empty_cache()
```

```
--- Measuring micro-iteration time for K=1 ---
K=1: mean micro-iteration time = 8.442 ms

--- Measuring micro-iteration time for K=2 ---
K=2: mean micro-iteration time = 10.359 ms

--- Measuring micro-iteration time for K=4 ---
K=4: mean micro-iteration time = 9.301 ms

--- Measuring micro-iteration time for K=8 ---
K=8: mean micro-iteration time = 8.988 ms
```

In [8]:
```
# Plot: Micro-iteration time vs K
plt.figure(figsize=(8, 5))
plt.plot(K_values, [micro_iter_times[k] for k in K_values], 'bo-', mar
plt.xlabel('Gradient Accumulation Factor K', fontsize=13)
plt.ylabel('Micro-Iteration Time (ms)', fontsize=13)
plt.title('Part 1: Micro-Iteration Time vs K', fontsize=14)
plt.xticks(K_values)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



Part 1: Your Answer

Explain why the micro-iteration time does or does not change with K in a single-GPU setup.

Although the graph is not perfectly flat, there is no consistent trend with increasing K. Each micro-iteration processes the same batch size (64) and

performs identical forward and backward computations, so the theoretical compute cost is independent of K. The small fluctuations observed are due to GPU scheduling, memory allocation patterns, and synchronization noise rather than changes in SGD behavior. Therefore, gradient accumulation does not materially affect micro-iteration time on a single GPU.

Part 2: Update Time and Effective Throughput [4 points]

Task: For each value of K, measure:

- Average *optimizer update time* (seconds per optimizer step), averaged over the middle 60% of updates (skip first 20% and last 20%).

The total number of updates U is computed per K as $U = 5 \times \lfloor 10240/(64 \times K) \rfloor$.

Then compute the *effective throughput* (images/sec):

$$\text{Throughput} = \frac{64 \times K}{T_{\text{update}}}$$

- Plot throughput vs. effective batch size ($64 \times K$)

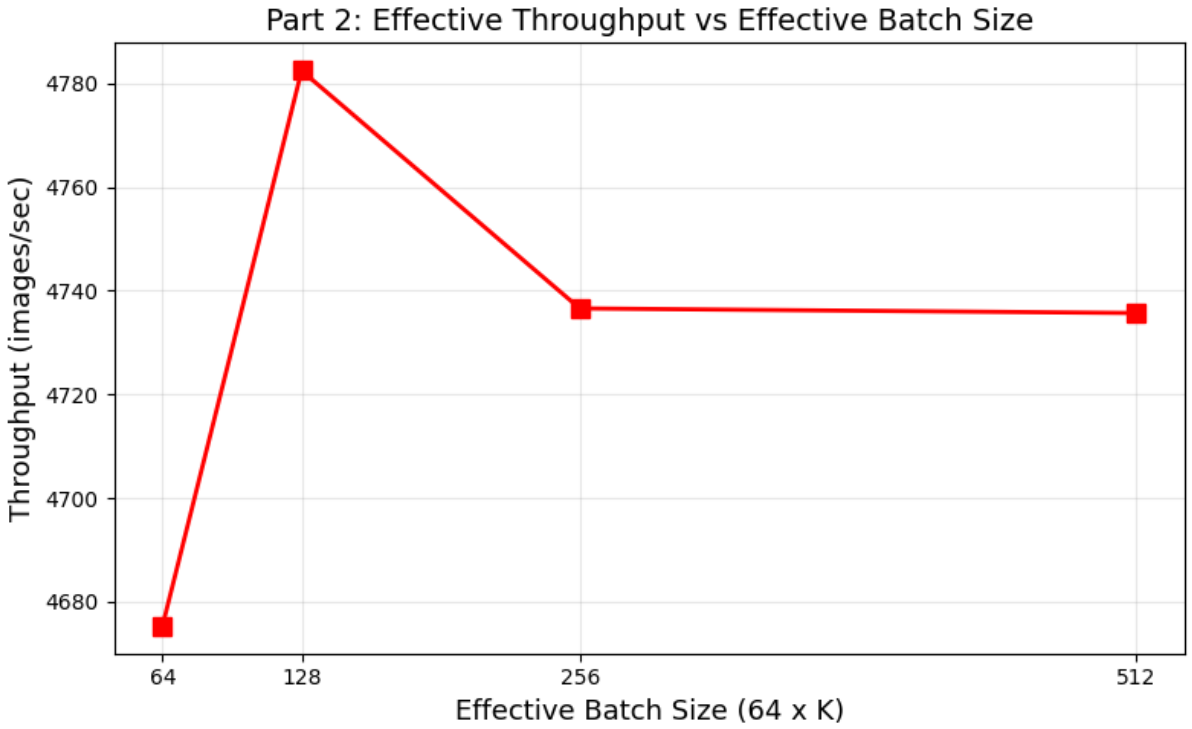
In [9]:
```
NUM_EPOCHS = 5
TRAIN_SIZE = len(train_subset)  # 10240

update_times = {}     # K -> mean update time in seconds
throughputs = {}       # K -> images/sec

for K in K_values:
    U = NUM_EPOCHS * (TRAIN_SIZE // (64 * K))
    skip = U // 5  # skip first/last 20%
    print(f"\n--- Measuring update time for K={K} (U={U}, middle windo
    set_seed(42)
    model = create_model()
    optimizer = create_optimizer(model)
    criterion = nn.CrossEntropyLoss()
    train_loader = create_train_loader()
    data_iter_ref = [iter(train_loader)]

    def get_batch():
        try:
            return next(data_iter_ref[0])
        except StopIteration:
```

```
            data_iter_ref[0] = iter(train_loader)
            return next(data_iter_ref[0])

    model.train()

    # TODO: For each of U optimizer updates:
    # - Use torch.cuda.synchronize() before start/end timing
    # - Time the full update (K micro-iterations + optimizer step)
    # - Each micro-iteration: get batch, move to device, forward, loss
    # - After K micro-iterations: optimizer.step()
    # - Record update time
    # Then: average over middle 60% of updates (skip first/last 20%)
    # Store mean_update_time in update_times[K]
    # Compute throughput = (64 * K) / mean_update_time and store in th

    times = []

    # Ensure clean grads
    optimizer.zero_grad(set_to_none=True)

    for u in range(U):
        # Time the FULL update: K micro-iterations + optimizer.step()
        torch.cuda.synchronize()
        t0 = time.perf_counter()

        optimizer.zero_grad(set_to_none=True)

        for k in range(K):
            images, labels = get_batch()
            images = images.to(device, non_blocking=True)
            labels = labels.to(device, non_blocking=True)

            outputs = model(images)
            loss = criterion(outputs, labels)

            # Average gradient over the effective batch
            (loss / K).backward()

        optimizer.step()

        torch.cuda.synchronize()
        t1 = time.perf_counter()

        times.append(t1 - t0)

    # Average over middle 60% of updates
    mid_times = times[skip:U - skip]
    mean_update_time = float(np.mean(mid_times))

    update_times[K] = mean_update_time
    throughputs[K] = (64 * K) / mean_update_time
```

```
    print(f"K={K}: mean update time (middle 60%) = {mean_update_time:.
    print(f"K={K}: throughput = {throughputs[K]:.2f} images/sec")

    del model, optimizer
    torch.cuda.empty_cache()
```

```
--- Measuring update time for K=1 (U=800, middle window: [160:640]) ---
K=1: mean update time (middle 60%) = 0.013689 s
K=1: throughput = 4675.38 images/sec

--- Measuring update time for K=2 (U=400, middle window: [80:320]) ---
K=2: mean update time (middle 60%) = 0.026765 s
K=2: throughput = 4782.43 images/sec

--- Measuring update time for K=4 (U=200, middle window: [40:160]) ---
K=4: mean update time (middle 60%) = 0.054047 s
K=4: throughput = 4736.58 images/sec

--- Measuring update time for K=8 (U=100, middle window: [20:80]) ---
K=8: mean update time (middle 60%) = 0.108115 s
K=8: throughput = 4735.69 images/sec
```

In [10]:
```
# Plot: Throughput vs Effective Batch Size
effective_batch_sizes = [64 * K for K in K_values]

plt.figure(figsize=(8, 5))
plt.plot(effective_batch_sizes, [throughputs[k] for k in K_values], 'r
plt.xlabel('Effective Batch Size (64 x K)', fontsize=13)
plt.ylabel('Throughput (images/sec)', fontsize=13)
plt.title('Part 2: Effective Throughput vs Effective Batch Size', font
plt.xticks(effective_batch_sizes)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



Part 2: Your Answer

Does throughput improve proportionally with K? Why or why not?

Throughput does **not** improve proportionally with K.

While increasing K increases the effective batch size, each optimizer update requires K forward and backward passes. Thus, update time grows approximately linearly with K:

$$T_{\text{update}}(K) \approx K \cdot T_{\text{micro}} + T_{\text{step}}$$

Although the optimizer step overhead is amortized as K increases (leading to a small improvement from K=1 to K=2), the dominant cost remains the K micro-iterations. As a result, throughput quickly saturates instead of scaling linearly with K.

This explains why the plot shows only a slight improvement rather than proportional scaling.

Part 3: Convergence Under a Fixed Epoch Budget [4 points]

Task: For each value of K, train for $E = 5$ epochs (so U varies by K) and

record:

- Training loss at each epoch boundary
- Test accuracy at the end of training

Plot training loss vs. epoch for all K on the same figure.

> All configurations see the same total data (51,200 images). Smaller K gets more optimizer updates.

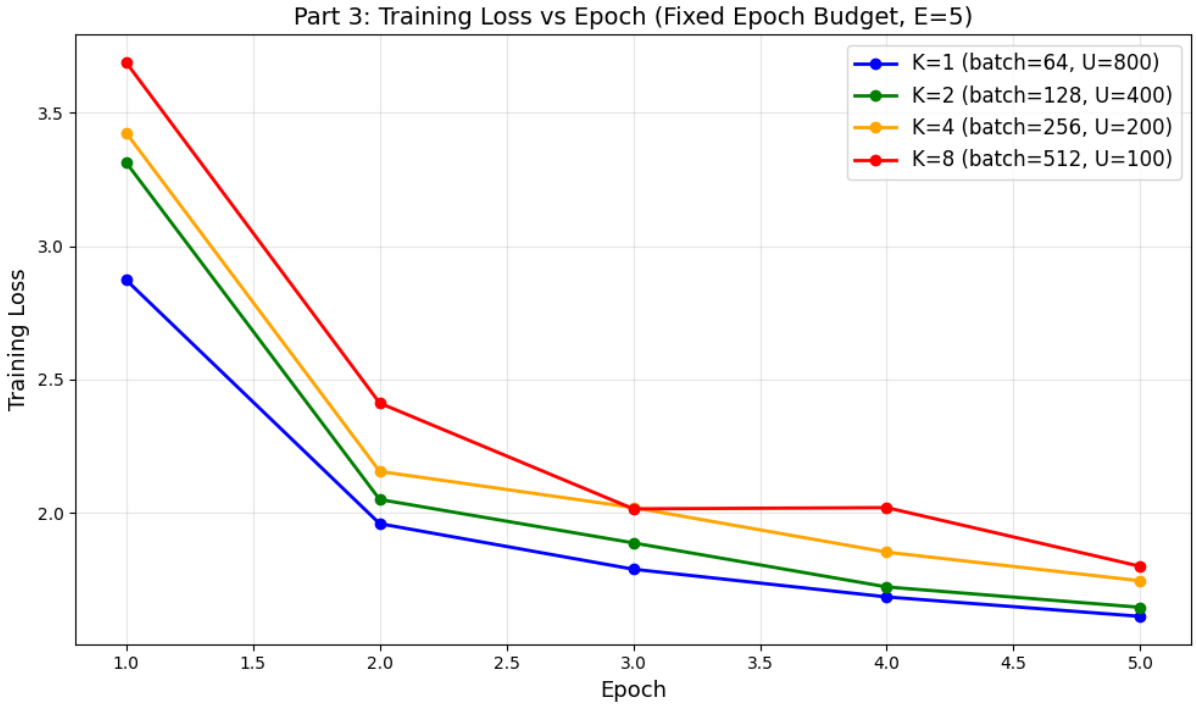
In [14]:
```
NUM_EPOCHS = 5
TRAIN_SIZE = len(train_subset)  # 10240

# Store results
all_losses = {}        # K -> list of (epoch, loss)
final_test_accs = {}   # K -> test accuracy

test_loader = create_test_loader()

for K in K_values:
    updates_per_epoch = TRAIN_SIZE // (64 * K)
    U = NUM_EPOCHS * updates_per_epoch
    print(f"\n{'='*60}")
    print(f"Training with K={K} (effective batch size = {64*K})")
    print(f"  Updates/epoch = {updates_per_epoch}, Total U = {U}")
    print(f"{'='*60}")

    set_seed(42)
    model = create_model()
    optimizer = create_optimizer(model)
    criterion = nn.CrossEntropyLoss()
    train_loader = create_train_loader()
    data_iter_ref = [iter(train_loader)]

    def get_batch():
        try:
            return next(data_iter_ref[0])
        except StopIteration:
            data_iter_ref[0] = iter(train_loader)
            return next(data_iter_ref[0])

    model.train()

    # TODO: Train for NUM_EPOCHS epochs, each with updates_per_epoch o
    # For each epoch:
    #   - Track running loss across all updates in the epoch
    #   - Each update: zero_grad, K micro-iterations with (loss/K).bac
    #   - Remember to move data to device with .to(device)
    #   - At end of epoch: compute average loss, append (epoch, avg_lo
    #   - Print epoch loss
```

```
    # After training: evaluate on test_loader, store in final_test_acc
    # Store losses_log in all_losses[K]

    losses_log = []

    for epoch in range(1, NUM_EPOCHS + 1):
        epoch_loss_sum = 0.0
        epoch_micro_batches = 0  # counts 64-sized micro-batches (shou

        for upd in range(updates_per_epoch):
            optimizer.zero_grad(set_to_none=True)

            # K micro-iterations per optimizer update
            for k in range(K):
                images, labels = get_batch()
                images = images.to(device, non_blocking=True)
                labels = labels.to(device, non_blocking=True)

                outputs = model(images)
                loss = criterion(outputs, labels)

                # accumulate averaged gradients
                (loss / K).backward()

                # log loss on the real (unscaled) micro-batch
                epoch_loss_sum += loss.item()
                epoch_micro_batches += 1

            optimizer.step()

        avg_epoch_loss = epoch_loss_sum / max(1, epoch_micro_batches)
        losses_log.append((epoch, avg_epoch_loss))
        print(f"Epoch {epoch}/{NUM_EPOCHS} | avg train loss = {avg_epo

    # Final evaluation
    test_acc = evaluate(model, test_loader)
    final_test_accs[K] = test_acc
    all_losses[K] = losses_log

    print(f"Final Test Acc (K={K}): {test_acc:.2f}%")

    del model, optimizer
    torch.cuda.empty_cache()
```

```
============================================================
Training with K=1 (effective batch size = 64)
  Updates/epoch = 160, Total U = 800
============================================================
Epoch 1/5 | avg train loss = 2.8729
Epoch 2/5 | avg train loss = 1.9601
Epoch 3/5 | avg train loss = 1.7896
Epoch 4/5 | avg train loss = 1.6857
Epoch 5/5 | avg train loss = 1.6129
Final Test Acc (K=1): 40.23%

============================================================
Training with K=2 (effective batch size = 128)
  Updates/epoch = 80, Total U = 400
============================================================
Epoch 1/5 | avg train loss = 3.3129
Epoch 2/5 | avg train loss = 2.0507
Epoch 3/5 | avg train loss = 1.8882
Epoch 4/5 | avg train loss = 1.7232
Epoch 5/5 | avg train loss = 1.6468
Final Test Acc (K=2): 40.23%

============================================================
Training with K=4 (effective batch size = 256)
  Updates/epoch = 40, Total U = 200
============================================================
Epoch 1/5 | avg train loss = 3.4241
Epoch 2/5 | avg train loss = 2.1558
Epoch 3/5 | avg train loss = 2.0206
Epoch 4/5 | avg train loss = 1.8533
Epoch 5/5 | avg train loss = 1.7462
Final Test Acc (K=4): 38.82%

============================================================
Training with K=8 (effective batch size = 512)
  Updates/epoch = 20, Total U = 100
============================================================
Epoch 1/5 | avg train loss = 3.6878
Epoch 2/5 | avg train loss = 2.4118
Epoch 3/5 | avg train loss = 2.0157
Epoch 4/5 | avg train loss = 2.0203
Epoch 5/5 | avg train loss = 1.8003
Final Test Acc (K=8): 36.18%
```

In [15]:
```python
# Plot: Training loss vs epoch for all K
plt.figure(figsize=(10, 6))
colors = ['blue', 'green', 'orange', 'red']

for K, color in zip(K_values, colors):
    epochs = [x[0] for x in all_losses[K]]
    losses = [x[1] for x in all_losses[K]]
    plt.plot(epochs, losses, f'-o', color=color, markersize=6, linewid
             label=f'K={K} (batch={64*K}, U={NUM_EPOCHS * (TRAIN_SIZE
```

```
plt.xlabel('Epoch', fontsize=13)
plt.ylabel('Training Loss', fontsize=13)
plt.title('Part 3: Training Loss vs Epoch (Fixed Epoch Budget, E=5)',
plt.legend(fontsize=12)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# Print final test accuracies
print("\nFinal Test Accuracies:")
for K in K_values:
    U = NUM_EPOCHS * (TRAIN_SIZE // (64 * K))
    print(f"  K={K} (batch={64*K}, U={U}): {final_test_accs[K]:.2f}%")
```




```
Final Test Accuracies:
  K=1 (batch=64, U=800): 40.23%
  K=2 (batch=128, U=400): 40.23%
  K=4 (batch=256, U=200): 38.82%
  K=8 (batch=512, U=100): 36.18%
```

Part 3: Your Answer

Discuss how increasing effective batch size affects convergence when all configurations train for the same number of epochs (same total data).

Increasing the effective batch size (larger K) slows convergence under a fixed epoch budget, even though all configurations see the same total number of training images.

From the plot, smaller K (e.g., $K = 1$) consistently achieves lower training loss

at every epoch, while larger K (e.g., $K = 8$) converges more slowly and ends with higher loss after 5 epochs.

The reason is that although all configurations process the same total number of images (51,200), smaller K performs more optimizer updates:

- Smaller batch $\rightarrow$ more updates $\rightarrow$ more frequent parameter adjustments
- Larger batch $\rightarrow$ fewer updates $\rightarrow$ slower parameter refinement

While larger batches provide lower-variance gradient estimates, they perform fewer update steps within the same number of epochs. As a result, convergence (in terms of training loss) is slower for larger effective batch sizes.

Required Summary Table

Run the cell below to generate the summary table with all measured values.

In [16]:
```
print("="*90)
print("SUMMARY TABLE")
print("="*90)
print(f"{'K':>3} | {'Eff. Batch':>10} | {'Total U':>8} | {'Micro-Iter
print("-"*90)
for K in K_values:
    U = NUM_EPOCHS * (TRAIN_SIZE // (64 * K))
    print(f"{K:>3} | {64*K:>10} | {U:>8} | {micro_iter_times[K]:>15.2f
print("="*90)
```

```
==========================================================================
==================
SUMMARY TABLE
==========================================================================
==================
  K | Eff. Batch |  Total U | Micro-Iter (ms) | Update Time (s) | Test
Acc (%)
--------------------------------------------------------------------------
------------------
  1 |         64 |      800 |            8.44 |          0.0137 |
40.23
  2 |        128 |      400 |           10.36 |          0.0268 |
40.23
  4 |        256 |      200 |            9.30 |          0.0540 |
38.82
  8 |        512 |      100 |            8.99 |          0.1081 |
36.18
==========================================================================
==================
```

Part 4: Connection to Real Synchronous SGD [3 points]

Answer the following questions in the cells below.

Q4a: Which aspects of synchronous SGD are captured by gradient accumulation?

Gradient accumulation captures the **algorithmic behavior** of synchronous SGD:

- It increases the effective global batch size to $64 \times K$.
- Gradients are averaged across K micro-batches before an update.
- Parameter updates occur using the same averaged gradient that would be produced by K workers in synchronous data parallel training.
- The optimization trajectory (for fixed batch size and learning rate) matches large-batch synchronous SGD in expectation.

Thus, gradient accumulation correctly simulates the mathematical update rule of synchronous SGD.

Q4b: Which system-level effects are missing?

Gradient accumulation does not capture system-level distributed effects such as:

- Inter-GPU communication (all-reduce) latency
- Communication bandwidth limits
- Synchronization barriers between workers
- Straggler effects (slowest worker determines step time)
- Network topology constraints (PCIe vs NVLink vs InfiniBand)

Since everything runs on a single GPU, there is no gradient communication overhead or synchronization delay.

Q4c: In real multi-GPU synchronous SGD, how would the missing effects change iteration time, scaling efficiency, and the optimal number of workers?

In real multi-GPU synchronous SGD:

$$T_{\text{iteration}} = T_{\text{compute}} + T_{\text{all-reduce}}$$

As the number of workers increases, communication time grows and can

eventually dominate computation time.

- Iteration time increases due to communication overhead.
- Scaling efficiency becomes sub-linear.
- Adding more workers yields diminishing returns.
- There exists an optimal number of workers that balances compute parallelism and communication cost.

Thus, real distributed training does not scale perfectly, unlike single-GPU gradient accumulation which ignores communication costs.

Notes

- The goal is to observe trends; minor differences across A100 vs. L4 are expected.
- All configurations train for E=5 epochs (51,200 images). The number of optimizer updates varies by K.
- Do not modify the dataset subsets, model, or hyperparameters.
- Keep your code deterministic where possible (random seeds are set above).