Instagram Post Analytics Predictor

1 User and Decision

Target user: Instagram content creators or brands wanting to maximize engagement (likes, comments, shares) on their posts.
Decision: Whether to post the content as it is, or make small adjustments such as to the timing of the post, photo content, or caption tone before publishing to improve engagement.

2 Target and Horizon

Target: A numeric engagement level such likes, comments, or shares on a post.
Timeline: Engagement within the first 24 hours of posting.
Type: Done by regression on weights to determine engagement score.

3 Features (No Leakage)

Post metadata: Caption sentiment analysis and length, posting time of day, hashtags Photo analysis of content: Content, People, Photo Quality. Account history, analysis of engagement of past posts, posting frequency, comment analysis and what people want to see.
Exclusions: Prediction time leakage: Future engagement data such as likes, comments, or information attained after posting. For example, after the user presses a post, the API should not use its current engagement stats to factor in improvements to the post. Training Time Leakage: During training, we cannot use any data that occurred after a training post was published in order to predict it because then it may suffer leakage. When deciding a training set, we must make sure future posts never leak backwards in training.

4 Baseline → Model Plan

Baseline: A rule-based model that returns the moving average of engagement (likes) for the user's last 5 posts can be a suitable minimal benchmark that would be simple to implement and require little amounts of data. The assumption here is that the user's current post will be similar to their recent posts in engagement. This can be helpful as future models should exceed the performance of this baseline as they will have more access to data.

5 Metrics, SLA, and Cost

Metrics: Since we are using regression, Mean Absolute Percent Error (MAPE) could be an appropriate way of measuring the degree of accuracy of the model in a human readable way. Since it reports things as percentages, it would be useful to see how well the model performs across different magnitudes (small content creators vs. larger content creators). However, for predictions with small engagement, predicting 10 likes vs. 1 like, then you can have crazy huge errors (900%). To counteract this, we will also track Mean Absolute Error (MAE) as a secondary metric, which is more stable for small number cases. SLA: p95 latency < 500ms per prediction request. Cost envelope = free-tier hitting 100 requests/day and should automatically scale at around $<$ \$200/day at 50k requests/hour spikes. (More benchmarking and research is required to get a more accurate representation of costs and max costs)

6 Minimal Evaluation Plan

Train our model while avoiding leakages (done by splitting recent posts as our test set, while keeping historical posts as our training set).
Use Mape as our primary metric to optimize for during training, and MAE as a potentially secondary metric. Then compare the trained model against our moving average baseline to see which performs better.
Measure the end-to-end request time from a client request to a response. Can be done by looking at the Networks tab on the browser developer console for some sample real world use case timings.
Test normal use cases like 100 req/day which should sufficiently meet our p95 $<$ 500ms. Do some spike testing for 50k req/hour for a short amount of time to see how the system reacts. Measuring the time taken for each request to calculate other p50/p95/p99.
For costs, estimate reads/writes per request against the Database with user information. As well as invocation counts + execution time + memory allocation in GB-seconds for each request which can be used to calculate actual costs for running each lambda request.

7 API Design + Architecture

- API Endpoint: **/predict** endpoint sits on AWS API Gateway and handles POST requests to initiate inference against our models via AWS Lambdas. Post requests are more secure than GET requests and also allows us to better structurally embed metadata like text and images represented in binary using the **multipart/form-data** data type.

  Sample Request Payload:

```
{
  "account_id": "acct_12345",
  "post": {
    "text": "Sunset at the pier #sunset #photography",
    "scheduled_time_utc": "2025-09-20T18:00:00Z",
    "hashtags": ["sunset","photography"],
    "location": {"lat":49.2827,"lon":-123.1207},
    "metadata": {"client":"web-v0.9","platform":"ios"}
  },
  "explain": true
}
```

  Sample 200 OK response:

```
{
  "model_version":"v1.3.0",
  "prediction": {
    "predicted_engagement": 123.4,
    "predicted_bucket":"medium"
  },
  "suggestions":[
   {"feature":"post_time","action":"move_to","value":"2025-09-20T19:00:00Z","impact_estimate_pct":12},
   {"feature":"caption_sentiment","action":"increase_positive","impact_estimate_pct":6}
  ],
  "diagnostics":{
    "latency_ms": 312,
    "feature_fetch_ms": 45,
```

```
    "inference_ms": 180,
  }
}
```

- Login and Authentication: Authentication will be performed with **Amazon Cognito** that will use the **OAuth2** protocol to link with the user's account and fetch data. Cognito stores identities and credentials and will provision the keys to access a user's specific model weights that have been derived from their account post history.

- Predictor API Endpoint Infrastructure: The two key infrastructures to consider are AWS Lambdas vs Fargate. For a prototype startup that handles only 100 requests a day, it makes sense to use Lambdas. As serverless environments that only run on demand when a user makes a request, this makes them very cost effective because we only pay for useful work. Since Fargate is a containerized environment, it incurs costs continuously to keep it alive. For a small startup, we will be paying for a lot of idle time which is a lot of wastage and will eat into our Free quota.

  However, the major argument for using Fargate is the high performance latency of a cold start for lambdas. ML models are often quite large (Ten to hundreds of megabytes), which can take quite a bit of time to load from a cold start and could potentially be a major bottleneck against our p95 latency target. Since Fargate is constantly kept warm, we would be able to cut out the cold-start latency. In addition, while baking models into lambda environments works for smaller less complex models, if we plan on supporting larger dynamic model loading from S3, each lambda instance may incur a cost where loading the models and keeping them in memory for containers might be more cost effective. This makes Fargate an enticing option to switch to once we build up a sufficient customer base to amortise the constant overhead and to scale to larger and more complex functionality in each individual request.

  In the event of a request spike when our product goes viral hitting 50,000 req/hour, the elasticity of lambdas are considerably greater than Fargate containers. Lambdas automatically scale with requests and AWS handles load balancing, making them very flexible. The only major concern being configurable concurrency limits which can be adjusted. Fargate Containers on the other hand require smart autoscaling rules and intelligent logging to detect surges and launch extra containers as a response. However, these containers often have significant startup times and can take up to minutes to be ready. This delay can cause request throttling and possibly dropped requests if the response is not agile enough. In addition, overreacting with containers can potentially become expensive as many containers sit idle, while underreacting can cause requests to be dropped. Seeing as this is the first time we are handling a viral spike, it may be unrealistic to have the perfect scaling protocol amides uncertain data readings. Thus as an overall pick, it makes sense to stick with AWS lambdas as our starting architecture, but strongly consider migrating to Fargate containers if perhaps users stick around after the viral spike as returning customers.

- Third bullet point

7) Privacy, Ethics, Reciprocity (PIA excerpt) Data inventory, purpose limitation, retention, access (link your PIA). Telemetry decision matrix (value vs invasiveness vs effort). Guardrails: k-anonymity, jitter/aggregation, opt-ins, disclosure. Reciprocity: value returned and to whom. 8) Architecture Sketch (1 diagram) Architecture: Serverless AWS Lambda functions are appropriate for scalability. However a major limit to the current approach is cold start + possibly heavy computation times where memory limits and time limits may come into play. This is why to start, our p95 latency will be relatively high, Lambdas are a good cost saving and efficient tool where we can run our tool, make the data base fetches, and run our model from a lambda and serve the result. However, as we scale, we may look towards tools like AWS Fargate to containerize our servers and decrease the latency of cold starts. Major components and data flow. Note trade-offs and alternatives. 9) Risks Mitigations Top 3 risks (technical/ethical) and how you will test or reduce them.