ChatGPT

Automated Pitch Generation: Evaluation Strategies and Prompt Evolution

Introduction

Generating persuasive and creative pitches at scale requires not only a strong generation mechanism but also robust evaluation and refinement loops. In the context of a community-focused Go programming group, pitches must appeal to both engineers and business professionals. They should be **persuasive, creative, community-oriented**, include **relevant statistics or recent examples**, and avoid off-target content (e.g. not promoting jobs outside the company). Manually iterating on each pitch is infeasible, so we need an automated system to **generate batches of pitch candidates, evaluate their quality, and evolve prompts** over time. This report explores strategies for automated pitch evaluation, scoring criteria, and prompt evolution techniques – including existing tools, models, and metrics – to streamline the workflow of selecting the best pitches for human review.

Evaluation Criteria for Pitches

Before choosing an evaluation method, it's important to define the dimensions of quality for a good pitch in this context. The following table outlines key criteria and their meanings:

Criterion	Description
Creativity	How original and engaging the pitch is. A creative pitch presents fresh ideas or analogies and avoids clichés, making it memorable.
Persuasiveness	The degree to which the pitch convinces and motivates the audience. This includes clear benefits, emotional or logical appeals, and a compelling call-to-action.
Clarity	How clear and easy to understand the pitch is for a mixed audience. The language should be concise and accessible to both engineers and business professionals, with minimal jargon and a logical flow.
Statistical Grounding	Use of relevant facts, numbers, or recent examples to support claims. A strong pitch might cite a statistic or success example (with a source) to add credibility.
Thematic Relevance	Alignment with the Go community focus and audience interests. The pitch should emphasize community benefits, Go language advantages, or success stories, and avoid irrelevant or disallowed themes (e.g. promoting external job opportunities).

Using these criteria, we can formulate a scoring rubric. Each pitch can be rated on each dimension (e.g. on a 1–5 scale), and an aggregate score or profile can be used to compare pitches. An example scoring rubric is shown below:

Dimension	Poor (1)	Excellent (5)
Creativity	Clichéd or generic phrasing; no unique angle.	Original, novel approach or story; stands out memorably.
Persuasiveness	Not convincing or motivating; lacks evidence or call-to-action.	Very compelling with clear benefits, emotional appeal, and a strong call-to-action.
Clarity	Confusing or jargon-heavy; hard for mixed audience.	Crystal clear and concise; easily understood by technical and non-technical readers.
Statistical Grounding	No real evidence or unsupported claims.	Backs claims with a relevant statistic or example (properly cited) that strengthens credibility.
Thematic Relevance	Off-topic or misaligned with community and Go focus.	Directly relevant to Go community interests; highlights community and technology benefits appropriately.

These criteria and rubrics can guide both human reviewers and automated evaluators. Next, we discuss how to implement automated evaluation against these dimensions.

Automated Evaluation Strategies

Automating the evaluation of pitch quality can be approached with a combination of techniques. No single metric suffices for complex, open-ended text, so we consider multiple strategies ranging from simple heuristics to advanced AI-based judges. Below we outline the main approaches and how they can be applied:

Rule-Based Heuristics

Rule-based heuristics involve explicit checks and simple metrics that can be computed automatically. While they may not capture nuance, they are easy to implement and can quickly filter out obviously poor pitches or flag specific issues. Examples of heuristics for our pitch scenario include:

- **Keyword and Content Checks:** Ensure the pitch mentions relevant keywords (e.g. "Go", "community", "engineers", "business") to gauge thematic relevance. Conversely, flag or penalize pitches that contain disallowed content (for example, mentioning "job opportunities elsewhere" if that's off-limits).
- **Length and Readability:** Check that the pitch length is reasonable (neither too short nor verbose) and compute a readability score (such as Flesch Reading Ease). A clear pitch should score well on readability for a broad audience.
- **Statistic Presence:** Verify the presence of at least one number, statistic, or recent example. For instance, a regex can detect percentages or years, indicating the pitch includes factual grounding. If the pitch JSON output includes a list of sources, the evaluator can ensure that at least one source is present (to satisfy **Statistical Grounding**) and perhaps even validate that the source URL/domain is reputable.

- **Sentiment or Tone Analysis:** Use a sentiment analysis tool or lexicon to check that the tone is positive and enthusiastic (appropriate for a motivational community pitch). A very negative or neutral tone might be less persuasive.
- **Jargon Count:** Count technical jargon terms – a high count might mean the pitch is too technical for business readers, hurting **Clarity**. If jargon is present, the heuristic could require that it's explained or kept minimal.

These heuristics can be combined into a simple rule-based scorer. For example, one could assign a point for each criterion satisfied (e.g. +1 if a statistic is present, +1 if reading level is within target, etc.) to get a rough score. Rule-based methods are **transparent and fast**, but they don't truly understand the content. They might miss subtleties of persuasiveness or creativity. Therefore, they are often best used in conjunction with more advanced evaluators as a first-pass filter or as inputs to a larger scoring formula.

LLM-as-a-Judge (Large Language Model Evaluators)

LLM-as-a-Judge refers to using a large language model to evaluate and score text based on specified criteria. This has emerged as a powerful approach to assess open-ended text quality, as LLMs can handle nuanced language evaluation tasks that traditional metrics struggle with [1] [2]. The idea is to provide the LLM with an **evaluation prompt** that describes the criteria (like the rubric above) and asks the model to rate a given pitch. For example, the prompt could say: *"You are an expert writing coach. Evaluate the following pitch on creativity, persuasiveness, clarity, use of statistics, and relevance to the Go community, on a scale of 1-5 for each. Provide a brief justification for each score."* The model (ideally a powerful one like GPT-4) will then output scores for each dimension and perhaps an overall assessment.

LLM judges can be used in two modes: **direct scoring** of a single output, or **pairwise comparison** of two outputs [3]. In direct scoring, the model gives an absolute rating as described. In pairwise mode, the model is given two pitches and asked which is better (or to choose on each criterion), which can be more reliable for ranking. LLM-based evaluation is flexible – it can incorporate any criteria described in natural language [4]. For instance, we can instruct the LLM to focus on whether the pitch is *persuasive to both technical and business audiences* or if it *contains recent data*. The LLM will use its trained knowledge and reasoning to assess these qualities.

Advantages: LLM evaluators handle subjective and high-level criteria very well. They can judge style, persuasiveness, and coherence in ways that hard rules or traditional metrics cannot [5] [2]. They are also far faster and cheaper than human evaluations at scale, while often correlating well with human judgment if carefully prompted [6]. In fact, using GPT-4 as a judge has been shown as a practical alternative to human reviewers for many tasks, achieving consistency close to human agreement levels [6].

Considerations: The quality of LLM evaluation depends on the prompt and the model. It's important to **align the LLM judge with your criteria** by clearly specifying the rubric or even giving example evaluations. Some frameworks like **G-Eval** have been proposed, which use GPT-4 with a chain-of-thought prompting method to produce more reliable evaluations [7] [8]. G-Eval, for example, was introduced as a better alternative to rigid metrics like BLEU/ROUGE for creative tasks, and it has high consistency across runs [9]. One can implement such an approach by instructing the model to first reason about each criterion (step-by-step) and then output a score, reducing random variation. There are open-source tools like **DeepEval** that make it easy to define LLM-as-a-judge criteria and get scores with minimal code [10].

Potential Pitfalls: LLM judges can have their own biases or limitations. They might be overly generous or too harsh if not calibrated. It's wise to occasionally sanity-check their scores against human judgment on a sample. Additionally, using a very large model for evaluation has cost implications and latency; however, since this is offline evaluation, it may be acceptable given the benefits. For faster evaluation, one could use a slightly smaller model fine-tuned for evaluation (discussed next) or use the API with a lower temperature to ensure more deterministic scoring.

Fine-Tuned Scoring Models

Another strategy is to use or develop a **fine-tuned model** that predicts quality scores. This could be a classifier or regression model trained on examples of pitches labeled with quality ratings (if such data is available or can be synthesized). In the context of RLHF (Reinforcement Learning from Human Feedback), for instance, companies train **reward models** to judge outputs – essentially specialized evaluators that predict a preference score for a given text. In our case, one could imagine fine-tuning a smaller language model or even a BERT-style model to evaluate pitches on the five criteria, if we had a dataset of pitches with human scores.

Trained evaluators can be very fast at runtime (much faster than a big LLM) and, once well-trained, they apply a consistent rubric. Research suggests that such trained metrics often align better with human preferences than raw automated metrics [11]. However, they come with challenges: obtaining a high-quality labeled dataset of pitches may be difficult, and a model trained in one domain might not generalize well without retraining [11]. They can also be **fragile** – sensitive to the distribution of training data or even exploitable by adversarial inputs [11].

If pursuing this route, one might leverage transfer learning. For example, use an existing fine-tuned model for evaluating persuasion or text quality, and then fine-tune it slightly on a small set of Go-community pitches you rate yourself. Another approach is to use the LLM (from the previous strategy) to generate a synthetic training set: have it label a large number of generated pitches, and then train a smaller model to mimic those scores (a form of distillation). This gives a cheaper runtime evaluator after an upfront cost. That said, maintaining such a model is more complex than using an LLM directly or rules, so many practitioners prefer the LLM-as-a-judge unless latency/cost at scale is a huge concern.

Untrained Metrics and Heuristic Scores

In addition to the above, there are classic **untrained metrics** that might provide partial insight into pitch quality. These include things like **language model perplexity** (how likely the text is under a baseline model – although a high-quality creative pitch might actually *surprise* a model, so perplexity is not necessarily correlated with quality), or **embedding-based similarity** to some references. If we had an ideal reference pitch or a set of high-quality pitches, we could use metrics like **BERTScore** or cosine similarity in embedding space to see how close a candidate is to the known good examples. However, in our case there isn't a single "reference" pitch to match; we value originality too. So these metrics are not primary, but could be used to detect outliers (e.g., a pitch that is too dissimilar from all others might be either very novel or completely off-topic – either way it's something to flag for review).

We could also compute a **diversity metric** if generating many pitches: e.g., measure the number of unique n-grams or use self-BLEU (which checks how similar each pitch is to the others). This doesn't score a pitch's quality per se, but helps ensure the batch has variety and that we don't accidentally pick multiple very similar pitches. In selection, one might down-rank pitches that are near-duplicates of a higher-scoring one, to maximize coverage of ideas.

Combining Multiple Evaluation Methods

For robust automated evaluation, it's common to combine the above methods. For instance, you might use an LLM judge to get a detailed score on each criterion, and also apply a few **hard rules** (like "if no statistic cited, cap the score" or "if pitch is longer than 300 words, lower the clarity score"). You could also ensemble different judges or metrics: for example, use two different LLM prompts (or two models) to score and average their results for stability. Since each method has strengths and weaknesses, a hybrid approach can balance them.

Example: You generate 20 pitches, then for each: calculate a heuristic score (0-10) from rule-based checks, and ask an LLM to provide scores for each of the 5 criteria. You could then form a composite score where the LLM's overall judgment is primary but you subtract points if the heuristic found any major issue (like no stats or off-topic). The result might be a more trustable ranking than any single method alone.

Batch Generation and Score-Based Selection Workflow

With evaluation tools in hand, the system can run in an automated loop to produce and filter pitches. A typical **batch generation and selection workflow** might look like this:

1. **Batch Pitch Generation:** Use the existing agent (the pydantic-ai + Codex based system) to generate a batch of pitch candidates. For example, you might run the agent N times with slight variations or randomness (temperature sampling) to get 10–20 different pitches. Each run yields a structured JSON output (containing the pitch text and cited sources). Ensure generation has some variability (e.g., temperature > 0) so that the pitches aren't all identical. You could also prompt the agent with slightly different angles or prompts if you want more diversity in this batch.
2. **Automated Evaluation:** For each generated pitch, apply the chosen evaluation strategy. This could mean running an **evaluation LLM prompt** that reads the pitch and returns scores for creativity, persuasiveness, etc., as described earlier. In parallel, you might run some **heuristic checks** (e.g., does it have a stat? length okay? any forbidden phrases?). The result of this step is that every pitch gets a set of scores or flags. For example, Pitch #3 might get {Creativity: 5, Persuasiveness: 4, Clarity: 5, Stat Grounding: 5, Relevance: 5, Overall: 5} if it's nearly perfect, whereas Pitch #7 might get lower scores or a flag that it failed a check.
3. **Scoring and Ranking:** Normalize or aggregate these evaluations into a single **quality score** for each pitch. If using a multi-criteria LLM judge, you might instruct it to give an overall score, or you can compute one (e.g., a weighted sum of the criteria scores, if some factors are deemed more important). It's useful to keep the breakdown though, in case you want to display why a pitch was chosen or to help in prompt refinement later. At this stage, you will have a ranked list of pitches from highest scoring to lowest.
4. **Selection of Top Candidates:** Select the top *k* pitches (say the top 2 or 3) to pass on for human review and refinement. This could be done by picking the highest scoring ones, **with some diversity consideration**. For example, if two of the top three are very similar or make the same point, you might drop one in favor of the next one that offers a different angle. (This is where having an embedding-based similarity check or simply some manual inspection can help ensure variety in the

final set.) The selection step might also enforce any business rules (e.g., if a pitch scored high but had a minor policy violation, maybe it's excluded unless it can be fixed).

5. **(Optional) Regeneration or Improvement:** If none of the pitches met a certain quality threshold, or if some had great elements but also flaws, the system could automatically iterate: for instance, take a mediocre pitch and prompt an LLM to revise it for clarity or add a statistic. However, such *automated revision* is another level of complexity – an easier approach is to simply generate a large batch initially so that at least a few are good. The truly automated self-improvement will come from evolving the prompt itself (next section) rather than fixing individual outputs on the fly.

This workflow can be orchestrated with the help of various tools. Since your system already uses a ReAct-style agent with search, you might integrate the evaluation as another tool or a subsequent step in code. For example, after generation, the agent (or a separate evaluator agent) could be invoked to run the evaluation prompt using the same or a different model. The outputs could be stored (perhaps using Pydantic models for the evaluation results as well) and then sorted.

Tooling Tip: *OpenAI Evals* (the evaluation framework) or similar libraries can facilitate this batch evaluation process. You can define a custom eval that uses an LLM to score each output according to your rubric, and run it on many samples systematically. There are also high-level frameworks like **DeepEval** (by Confident AI) which implement LLM-as-a-judge patterns and provide consistency measures. These tools allow you to write a few lines specifying the criteria and then handle prompting and scoring across batches. Alternatively, you can script the loop manually using the OpenAI API or your LLM backend: generate N outputs, feed each into an evaluation prompt, and collect the scores.

By automating the generation and scoring in batches, you achieve a sort of *filtering pipeline*: a large variety of pitches is narrowed down to the best few, with quantitative justifications. This greatly speeds up the creative process, leaving humans to only fine-tune the top ideas rather than sift through everything. The next challenge is to **close the loop** by learning from these evaluations to improve future generations – i.e., evolving the prompt or generation strategy for even better pitches over time.

Prompt Evolution and Optimization Strategies

Over successive batches, we ideally want the pitch generator to improve, producing higher-quality outputs more consistently. Prompt evolution refers to systematically updating or optimizing the prompt (the instructions given to the LLM) based on the feedback (scores) from prior generations. Rather than relying on manual prompt engineering tweaks, we can use automated strategies to refine the prompt. Here are several approaches and tools for prompt evolution:

Evolutionary Algorithms for Prompt Optimization

One effective approach is to treat the prompt itself as something that can be optimized via an evolutionary algorithm (EA). In this paradigm, you maintain a **population of prompt variants** and evaluate how well each prompt performs (by generating pitches and scoring them). Then you select the best prompts and **mutate or recombine** them to create new prompt variants, iteratively improving the quality of outputs. Research in this area has shown promising results. For example, **EvoPrompt** is a framework that connects LLMs with evolutionary algorithms to optimize prompts, and it significantly outperformed manually crafted prompts on a variety of tasks [12] . EvoPrompt starts with a population of prompts and uses the LLM itself to generate new candidate prompts (applying mutation and crossover operations in natural language) and keeps the ones that lead to better performance on a development set [12]. Similarly, a recent approach called **GAAPO** (Genetic Algorithm Applied to Prompt Optimization) evolves prompts through successive generations using genetic algorithm principles [13]. GAAPO goes beyond simple random mutations by integrating specialized prompt generation strategies in the evolutionary loop [14] – for instance, some strategies might rephrase sentences, others might adjust the level of detail, etc., to see what yields the best results.

To apply this in our context, we could do the following: maintain, say, 5 different prompt phrasings for the pitch-generation agent. In each "generation," use each prompt to produce a few pitches, use the automated evaluator to score those pitches, and then assign a fitness score to the prompt (perhaps the average or maximum pitch score it produced). Then **select** the top-performing prompts (e.g. top 2) and generate new variants by either crossing them (mixing elements from both) or mutating them. Mutation could be as simple as rewording part of the prompt: for example, if the prompt currently says "Include a statistic to support your point", a mutation might try "Include a recent real-world statistic or study result". You can use the LLM (in a Codex/code prompt or text prompt) to help generate these mutations – essentially instruct the LLM to produce a slight variation of the prompt text, or use templates to swap in/out certain phrases. Over multiple generations, this evolutionary search may discover a prompt that consistently yields higher-scoring pitches. The cited research reports substantial improvements from such automated prompt optimization (up to 25% improvement on certain benchmark tasks by EvoPrompt) [15].

Prompt Mutation and Recombination Techniques

Even outside a full genetic algorithm framework, you can incorporate **prompt mutation** heuristics in a simpler way. Some ideas:

- **Parameter Sweeps:** Systematically vary certain prompt parameters or phrasing to see impact. For example, test versions of the prompt that emphasize different aspects: one variant might stress creativity ("be bold and think outside the box"), another stresses evidence ("back up claims with data"), etc. Evaluate which emphasis yields better pitches on average, and then refine accordingly.
- **A/B Testing Prompts:** In each batch generation, you could randomly pick between a few prompt versions for each run and then compare the distributions of scores. Over time, drop the poorer performing prompt versions and introduce new ones inspired by the better ones.
- **Recombining Successful Elements:** If one prompt variant led to very clear pitches and another led to very persuasive pitches, try to **combine their instructions**. For instance, one prompt might have an effective phrase like "use a storytelling approach", while another says "highlight a statistic from recent tech reports". Merging these into a single prompt could produce pitches with both qualities.
- **Auto-Refinement via LLM:** You can also use an LLM in a reflective way: after getting pitch outputs, ask the LLM (or another agent) to suggest how the prompt could be improved. For example: *"Given that the following pitch lacked a strong call-to-action, how should we modify the prompt to encourage that?"* The model might answer with a revised instruction. This is a form of AI-assisted prompt editing, which could be integrated into the loop occasionally to generate fresh prompt ideas.

Tools and Configuration for Automated Evolution

Implementing prompt evolution requires keeping track of multiple prompts and their performance. This is where some tooling can help: since you are using **pydantic-ai**, you could define a data structure for a PromptConfig that includes fields like the prompt text, its generation history, and performance stats. The system could save these to a JSON or database after each iteration. Using the Codex model (or any LLM that can execute code), you might even automate the orchestration: for example, a script (or agent tool) that modifies a prompt string in your config file or Python object and then triggers the next round of generation.

Projects like EvoPrompt and GAAPO have open-source code (for instance, Microsoft has released code for EvoPrompt [15]) which could be adapted or referenced for your use case. If integrating an existing library is feasible, it might accelerate development – otherwise a custom implementation with your agent framework is possible. Another relevant concept is **Promptbreeder** (a self-referential prompt evolution approach [16]), though it goes into research territory; the key idea is that the system itself can propose new prompts.

In practice, you might start with a simpler semi-automated approach: periodically review which prompts produced the top results and manually craft a new variant to test. But since the goal is full automation, gradually increasing the sophistication of the prompt search makes sense. Ensure that after each evolution step, you validate that the new prompt isn't overfitting to some quirk (for example, if the evaluation model has a bias, the prompt might exploit it to get a high score but produce content that a human would find strange). Keeping a human in the loop at major checkpoints (e.g. reviewing the highest-scoring pitch and the prompt that generated it every few iterations) can prevent degenerate solutions.

Conclusion

In summary, building an automated pipeline for pitch generation involves two synergistic components: **automated evaluation** to assess quality, and **prompt evolution** to refine the generation process. By defining clear criteria (creativity, persuasiveness, clarity, etc.) and leveraging LLM-based evaluators [4] (augmented with heuristics and possibly fine-tuned models), we can score large batches of candidate pitches quickly and with reasonable alignment to human preferences. From there, a score-driven selection process picks out the best pitches for human consideration, ensuring that only the most promising, well-crafted messages move forward.

To further enhance the system, automated prompt optimization techniques like evolutionary algorithms can be applied. These use evaluation feedback as a fitness signal to iteratively improve the prompt itself [12] [13], thus boosting the quality of future pitch generations. Modern tooling such as OpenAI's evaluation framework, DeepEval for LLM judges, and frameworks like Pydantic-AI or LangChain for agent orchestration can greatly assist in implementing these ideas. By combining these strategies, the agent will progressively learn to produce pitches that are not only high-quality by some abstract metric, but truly effective and resonant with the Go community audience – all with minimal manual tweaking.

Ultimately, this approach allows your team to **focus creative energy on reviewing and fine-tuning the top pitches**, while the heavy lifting of generation and preliminary curation is handled by the AI and its automated evaluators. The result is a scalable, adaptive pitch-writing assistant that continuously gets better at hitting the mark for your community-focused goals.

Sources: The strategies discussed are informed by recent research and practical guides on LLM evaluation and prompt optimization. For instance, using large language models as evaluators of generated text has been shown to be a flexible and effective alternative to manual or reference-based metrics [4] [2]. Techniques like G-Eval provide structured ways to prompt LLM judges for reliable scoring [8]. On the prompt optimization side, evolutionary methods (EvoPrompt, GAAPO) demonstrate how iterative prompt tweaking can outperform static human-written prompts [12] [14]. These insights, combined with tool support (OpenAI Evals, DeepEval, etc.), form a strong foundation for building the described pitch generation agent. Each component – generation, evaluation, and evolution – reinforces the others in a closed loop, enabling continuous improvement with minimal human intervention.

---

[1] [3] [4] [5] LLM-as-a-judge: a complete guide to using LLMs for evaluations
https://www.evidentlyai.com/llm-guide/llm-as-a-judge

[2] [6] Evaluating the Effectiveness of LLM-Evaluators (aka LLM-as-Judge)
https://eugeneyan.com/writing/llm-evaluators/

[7] [8] [9] [10] The Ultimate G-Eval Guide: LLM-as-a-Judge Debunked - Confident AI
https://www.confident-ai.com/blog/g-eval-the-definitive-guide

[11] A Unified Model for Automated Evaluation of Text Generation Systems
https://ercim-news.ercim.eu/en136/special/a-unified-model-for-automated-evaluation-of-text-generation-systems

[12] [15] [2309.08532] EvoPrompt: Connecting LLMs with Evolutionary Algorithms Yields Powerful Prompt Optimizers
https://arxiv.org/abs/2309.08532

[13] [14] GAAPO: Genetic Algorithmic Applied to Prompt Optimization
https://arxiv.org/html/2504.07157v3

[16] Self-Referential Self-Improvement via Prompt Evolution | OpenReview
https://openreview.net/forum?id=HKkiX32Zw1