Scalable Prompt Optimization in Apache Beam LLM Workflows

Tomi Ajakaiye

BEAM SUMMIT NYC 2025

Presenter Profile

Tomi Ajakaiye is an AI and data engineering consultant with over 10 years of experience managing and building large-scale, enterprise-grade business applications. She specializes in designing and deploying scalable solutions powered by large language models (LLMs) across different verticals and business domains such healthcare, finance, and government sectors.

As the Lead Partner at BluePod Consulting Inc., an AI consulting and data analytics firm based in Ontario, Canada, Tomi leads cross-functional teams in delivering innovative, secure, and compliant AI systems that drive operational efficiency and digital transformation.

Agenda

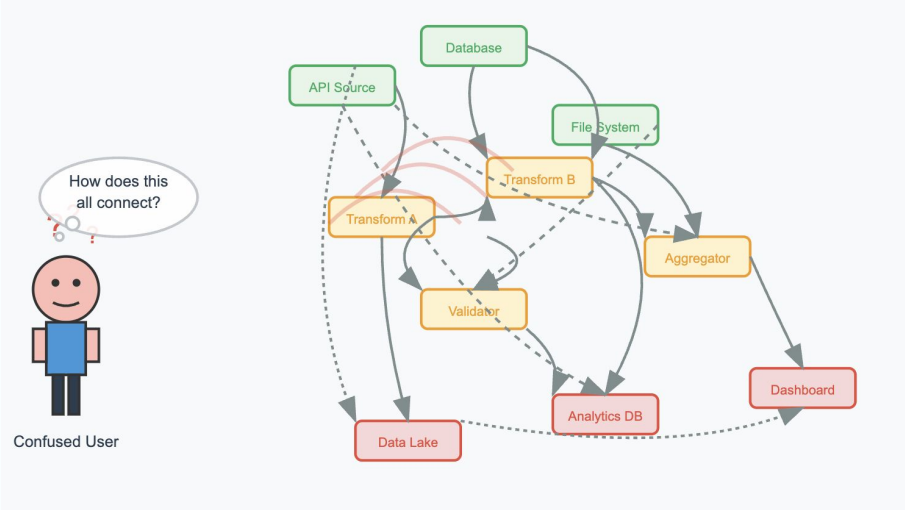
1. **The Pipeline Proliferation Problem:** When you have a pipeline for everything, how do you empower users to easily find and run the right one for their task?

2. **Our Vision: Intent-Driven Data Processing:** Moving beyond manual selection to a new paradigm where a user simply states their goal and the system takes care of the rest.

3. **The Pipeline Catalog: A Toolkit of Beam YAML Templates:** How we build and maintain a set of standardized, reusable, and highly optimized Beam YAML pipelines ready for execution.

4. **Simplicity, Governance, and the Future:** Discussing the powerful benefits of this approach and what's next for our intent-driven platform.

1. The Pipeline Proliferation Problem



A Victim of Our Own Success:

We love Apache Beam YAML Pipelines. Its power and flexibility allowed us to build robust, scalable pipelines for dozens of critical business tasks

A Pipeline for Everything:

We created highly-optimized, specialized pipelines for:

- Real-time sentiment analysis
- PII redaction in customer feedback
- Summarizing news articles
- Converting data formats for archival

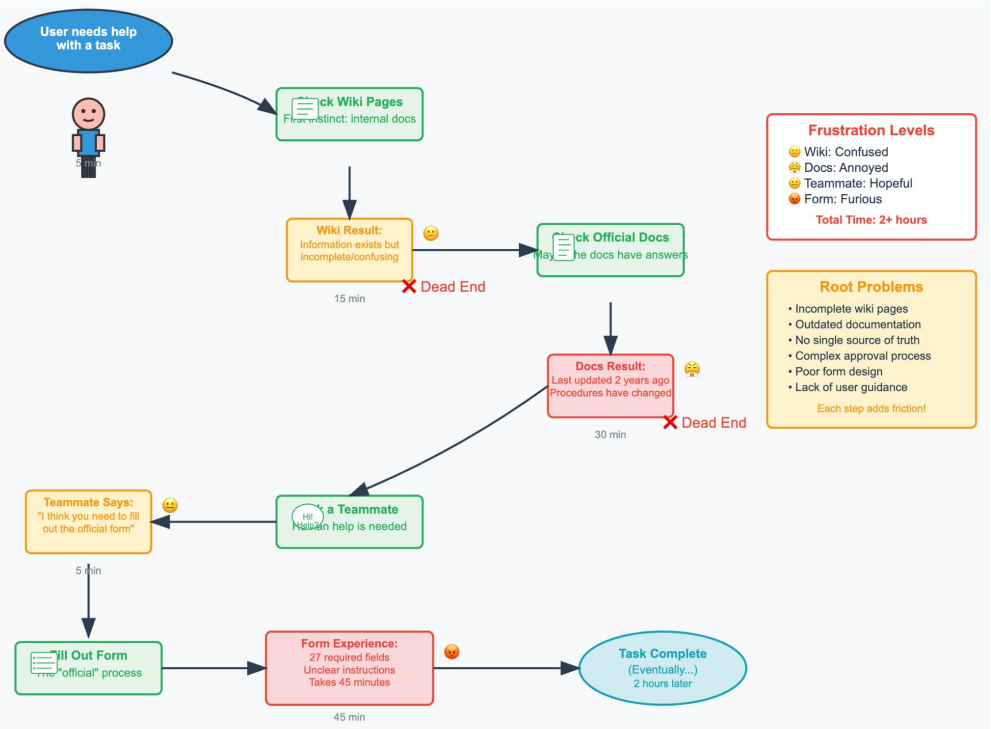
This specialization is great for efficiency and performance.

The Emerging Challenge:

- This success created a new kind of complexity. With so many tools available, a critical question emerged:
- How does a user, especially a non-technical one, know which pipeline to use and when?

1. The Pipeline Proliferation Problem

The Friction of Finding the Right Tool



The Documentation Trap:

Our first instinct was to create documentation: wikis, READMEs, and decision trees to guide users.

This solution quickly became the new problem.

Why Documentation Fails at Scale:

Cognitive Overload: Users face a wall of choices. Is my task "summarization" or "entity extraction"? Do I need the streaming or batch version?

Instant Obsolescence: Documentation is outdated the moment a new pipeline is added or modified. It's a constant source of maintenance debt.

The "Human Router" Bottleneck: We created a dependency on a few key engineers who became the only ones who knew how to run the right job for the right task.

The Beam YAML Paradox:

A user is still confronted with a list of files like `sentiment_v2.yaml`, `summarize_news.yaml`, and `redact_pii_streaming.yaml` and has no clear path to making the right choice

1. The Pipeline Proliferation Problem

The True Cost of Complexity

Lost Productivity
Valuable time is wasted hunting for the right tool instead of getting work done. Simple tasks that should take minutes can stretch for hours or even days.

Stifled Innovation
A product manager with a brilliant idea for a new data insight will simply give up if the barrier to entry is too high

Increased Risk of Error
Users are more likely to run the wrong pipeline, use incorrect parameters, or process data in a way that yields inaccurate or costly results.

The Path Forward

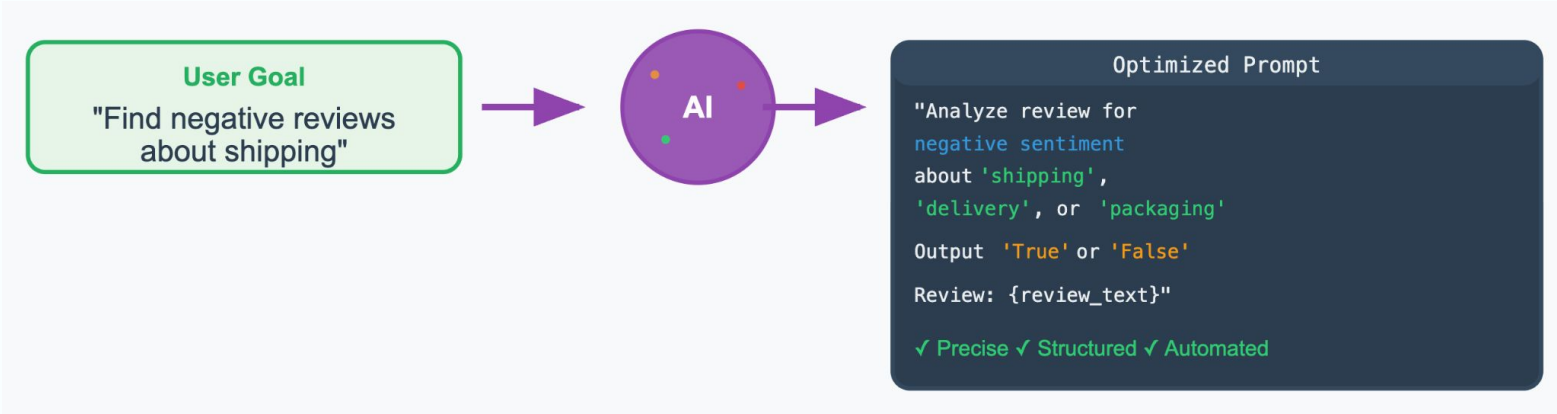
- We realized we needed to fundamentally change our approach. We had to move from a system where the **user finds the pipeline**, to a system where the **pipeline finds the user**.
- And that's what led us to our intent-driven solution.

New User Experience

A business user states their goal: *"Find negative reviews about shipping."*

The system generates an optimized prompt for the sentiment analysis pipeline:

```
"Analyze the following review for negative sentiment specifically related to 'shipping', 'delivery', or 'packaging'. Output 'True' if found, otherwise 'False'. Review: {review_text}"
```



2. Intent Driven Data Pipelines

1. The Beam YAML Pipeline Repository:

- This is our foundation. We maintain a version-controlled library of expert-built, reusable Beam pipelines. Each YAML file defines a specific capability (e.g., sentiment, summarization) and is designed to accept a prompt as a parameter.

2. The LLM-Powered Prompt Assistant:

- This is the core of our implementation. When a user selects a pipeline and states their goal, this LLM-based service acts as a co-pilot.
- It understands the requirements of the chosen Beam YAML pipeline and transforms the user's simple goal into a structured, optimized prompt query that will yield the best results from the model inside the pipeline.

3. Parameterized Pipeline Execution:

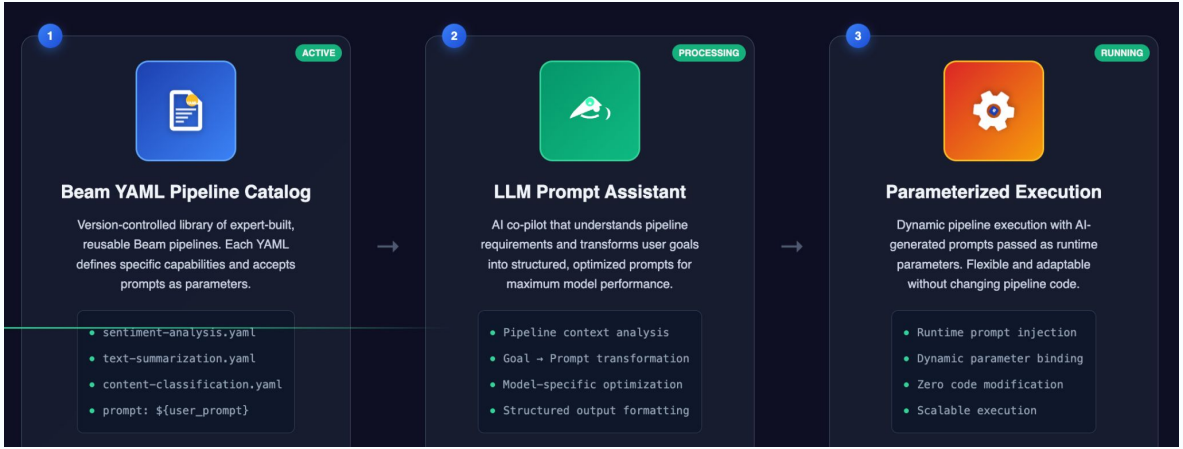
- The generated prompt isn't hard-coded. It's passed as a parameter at runtime when the Beam YAML pipeline is executed. This makes our pipelines flexible and dynamically adaptable to a wide range of user goals without ever changing the underlying pipeline code.

2. Intent Driven Data Pipelines

```yaml
pipeline:
  name: sentiment-analysis-pipeline
  description: "Parameterized sentiment analysis pipeline using BEAM"

  # Runtime parameters
  parameters:
    - name: analysis_prompt
      type: string
      description: "The prompt template for sentiment analysis"
      required: true
    - name: input_topic
      type: string
      description: "Input PubSub topic"
      required: true
    - name: output_table
      type: string
      description: "Output BigQuery table"
      required: true
    - name: model_endpoint
      type: string
      description: "ML model API endpoint"
      required: true

  # Pipeline transforms
  transforms:
    - name: ReadMessages
      type: ReadFromPubSub
      config:
        topic: "${input_topic}"

    - name: ParseJson
      type: Map
      config:
        function: |
          def parse_json(element):
            import json
            data = json.loads(element.decode('utf-8'))
            return {
              'id': data.get('id', ''),
              'text': data.get('text', ''),
              'timestamp': data.get('timestamp', '')
            }
```

```yaml
    - name: AnalyzeSentiment
      type: Map
      config:
        function: |
          def analyze_sentiment(message):
            import requests
            import json

            # Use parameterized prompt
            prompt = "${analysis_prompt}".format(text=message['text'])

            payload = {
              'prompt': prompt,
              'max_tokens': 100,
              'temperature': 0.1
            }

            response = requests.post(
              "${model_endpoint}",
              json=payload,
              headers={'Content-Type': 'application/json'}
            )

            if response.status_code == 200:
              result = response.json()
              return {
                'id': message['id'],
                'text': message['text'],
                'sentiment': result.get('sentiment', 'neutral'),
                'confidence': result.get('confidence', 0.0),
                'timestamp': message['timestamp']
              }
            else:
              return {
                'id': message['id'],
                'text': message['text'],
                'sentiment': 'error',
                'confidence': 0.0,
                'timestamp': message['timestamp']
              }
```

3. The Pipeline Repository: A toolkit for Beam YAML templates



Building a Standardized Toolkit

- We created a central repository for all our Beam YAML pipelines.
- Each pipeline is designed to be a modular "skill" (e.g., sentiment analysis, PII redaction).
- They are not just code; they are standardized assets.

The rules of the Catalog

- **Clear Naming & Versioning:** sentiment-analysis.v1.yaml is instantly understandable. Versioning in the filename and in Git allows for safe, iterative improvements.
- **Defined Interface:** Every pipeline in the repository clearly defines its expected inputs (e.g., a text field named user_review) and outputs.
- **Parameterization is Key:** Most importantly, each pipeline is designed to be configured at runtime, especially with the LLM prompt.

3. The Pipeline Repository: A toolkit for Beam YAML templates

The Pipeline Repository: Our Foundation for Scalability

To make our vision work, we had to stop treating pipelines as one-off scripts. Our core philosophy is: **Pipelines are products, not projects.**

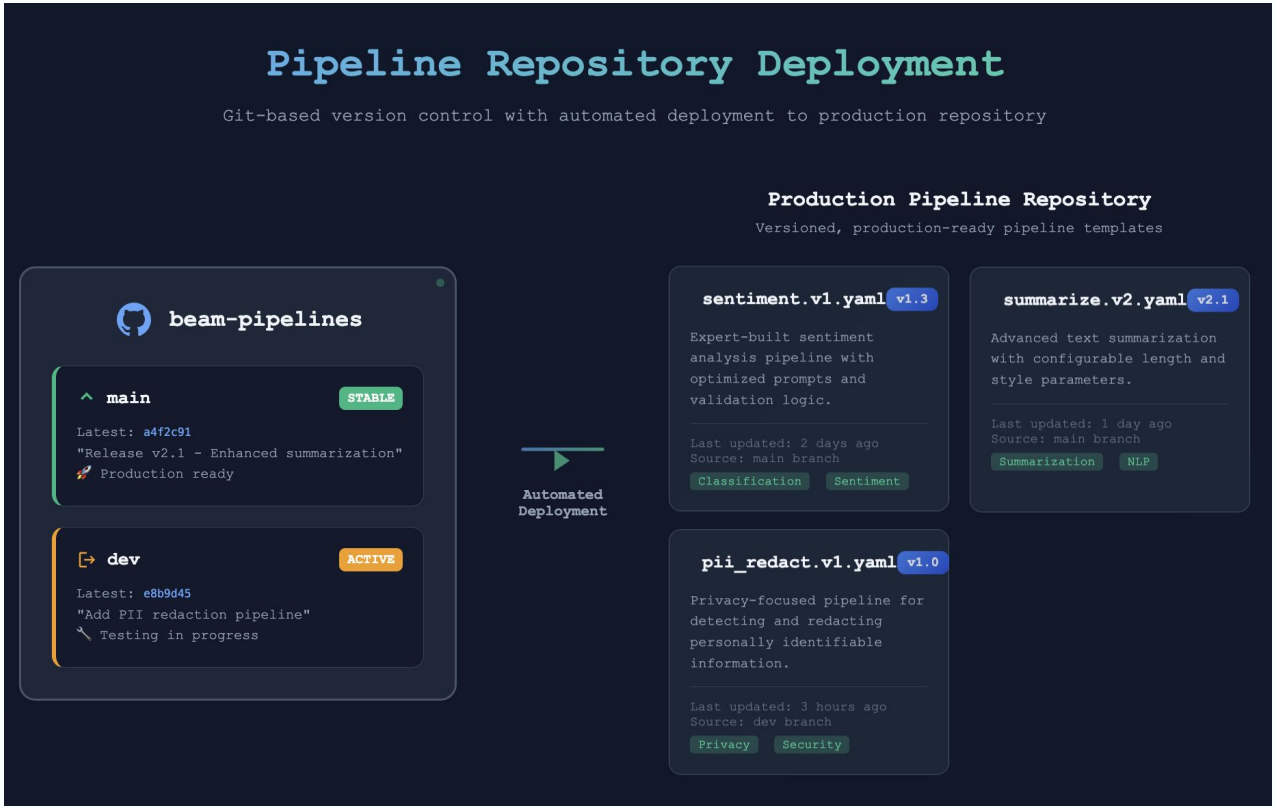
How We Build the Catalog:

- **Centralized & Version Controlled:** Every pipeline lives in a single Git repository. This is our single source of truth, allowing for branching, code reviews, and safe, iterative development.
- **Standardized Structure:** Each YAML file follows a strict template. This consistency means anyone on the team can understand a new pipeline's structure at a glance.
- **Designed for Reusability:** We don't build pipelines for a single use case. We build modular "skills"—like sentiment analysis or summarization—that can be applied to many different data sources and goals.

How We Maintain the Catalog:

- **Clear Naming Conventions:** A file is always named [skill].[version].yaml. There is never ambiguity about what a pipeline does or which version is being used.
- **Separation of Concerns:** The pipeline logic (the YAML) is kept completely separate from the application logic (the prompt). This allows us to update one without breaking the other.

3. The Pipeline Repository: A toolkit for Beam YAML templates



Key Features:

- **Goal-to-Prompt Translation**: User says "analyze customer feedback for app updates" → LLM generates optimized sentiment analysis prompt
- **Pipeline Intelligence**: Understands requirements for different pipeline types
- **Context Awareness**: Incorporates domain knowledge and specific use cases
- **Automatic Pipeline Selection**: Suggests best pipeline based on user description

LLM Prompt Assistant

```python
class LLMPromptAssistant:
    """
    LLM-powered prompt assistant that transforms user goals into optimized
    prompts for BEAM YAML pipelines
    """

    def __init__(self, api_key: str):
        self.client = openai.OpenAI(api_key=api_key)
        self.pipeline_registry = self._load_pipeline_registry()

    def _load_pipeline_registry(self) -> Dict[PipelineType, PipelineMetadata]:
        """Load metadata for all available BEAM pipelines"""
        return {
            PipelineType.SENTIMENT_ANALYSIS: PipelineMetadata(
                name="Sentiment Analysis Pipeline",
                description="Analyzes emotional tone of text data",
                input_format="JSON with 'text' field",
                output_format="sentiment classification with confidence",
                prompt_requirements=[
                    "Clear sentiment categories (positive/negative/neutral)",
                    "Confidence scoring mechanism",
                    "Handling of ambiguous cases",
                    "Context consideration instructions"
                ],
                optimization_tips=[
                    "Use specific examples for edge cases",
                    "Define confidence thresholds clearly",
                    "Consider domain-specific language",
                    "Handle sarcasm and irony explicitly"
                ]
            ),
            PipelineType.TEXT_CLASSIFICATION: PipelineMetadata(
                name="Text Classification Pipeline",
                description="Categorizes text into predefined classes",
                input_format="JSON with 'text' field",
                output_format="class label with confidence score",
                prompt_requirements=[
                    "Clear class definitions",
                    "Decision boundaries between classes",
                    "Handling of multi-class scenarios",
                    "Unknown/other category handling"
                ],
                optimization_tips=[
                    "Provide class descriptions with examples",
                    "Use hierarchical classification if needed",
                    "Define overlap resolution strategies",
                    "Include confidence calibration"
                ]
            ),
```

Generate Optimized Prompt

```python
def generate_optimized_prompt(self,
                              pipeline_type: PipelineType,
                              user_goal: str,
                              context: Dict[str, Any] = None) -> str:
    """
    Transform user goal into optimized prompt for the selected pipeline
    """
    pipeline_meta = self.pipeline_registry[pipeline_type]
    context = context or {}

    # Create system message for the LLM assistant
    system_message = f"""
    You are an expert prompt engineer specializing in {pipeline_meta.name}.

    Pipeline Details:
    - Purpose: {pipeline_meta.description}
    - Input Format: {pipeline_meta.input_format}
    - Output Format: {pipeline_meta.output_format}

    Requirements for effective prompts:
    {chr(10).join(f"- {req}" for req in pipeline_meta.prompt_requirements)}

    Optimization Guidelines:
    {chr(10).join(f"- {tip}" for tip in pipeline_meta.optimization_tips)}

    Your task: Transform the user's goal into a structured, optimized prompt that will yield the best results from the model

    Return ONLY the optimized prompt text, ready to be used in the pipeline.
    """

    # Create user message with goal and context
    user_message = f"""
    User Goal: {user_goal}

    Additional Context:
    {json.dumps(context, indent=2) if context else "None provided"}

    Generate an optimized prompt for this pipeline that addresses the user's goal.
    """

    # Call LLM to generate optimized prompt
    response = self.client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": system_message},
            {"role": "user", "content": user_message}
        ],
        temperature=0.1,
        max_tokens=1000
    )

    return response.choices[0].message.content.strip()
```

Pipeline Suggestion

```python
def get_pipeline_suggestions(self, user_description: str) -> List[PipelineType]:
    """
    Suggest appropriate pipelines based on user description
    """
    suggestion_prompt = f"""
    Based on this user description: "{user_description}"

    Available pipelines:
    {chr(10).join(f"- {pt.value}: {meta.description}" for pt, meta in self.pipeline_registry.items())}

    Which pipeline(s) would be most appropriate? Return only the pipeline names as a comma-separated list.
    """

    response = self.client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": suggestion_prompt}],
        temperature=0.1,
        max_tokens=200
    )

    suggested_names = [name.strip() for name in response.choices[0].message.content.split(',')]
    return [PipelineType(name) for name in suggested_names if name in [pt.value for pt in PipelineType]]
```

Beam Pipeline Orchestration

```python
class BeamPipelineOrchestrator:
    """
    Orchestrates the entire flow from user goal to pipeline execution
    """

    def __init__(self, prompt_assistant: LLMPromptAssistant):
        self.prompt_assistant = prompt_assistant

    def execute_pipeline_with_goal(self,
                                   user_goal: str,
                                   pipeline_type: PipelineType = None,
                                   context: Dict[str, Any] = None) -> Dict[str, Any]:
        """
        Complete flow: user goal → optimized prompt → pipeline execution
        """

        # Step 1: Pipeline selection (if not provided)
        if pipeline_type is None:
            suggested_pipelines = self.prompt_assistant.get_pipeline_suggestions(user_goal)
            if not suggested_pipelines:
                raise ValueError("No suitable pipeline found for the given goal")
            pipeline_type = suggested_pipelines[0]  # Use first suggestion

        # Step 2: Generate optimized prompt
        optimized_prompt = self.prompt_assistant.generate_optimized_prompt(
            pipeline_type, user_goal, context
        )

        # Step 3: Execute BEAM pipeline with optimized prompt
        pipeline_config = {
            "prompt": optimized_prompt,
            "input_topic": context.get("input_topic", "projects/default/topics/input"),
            "output_table": context.get("output_table", "default:results.output")
        }

        # This would trigger the actual BEAM pipeline execution
        execution_result = self._execute_beam_pipeline(pipeline_type, pipeline_config)

        return {
            "pipeline_type": pipeline_type.value,
            "user_goal": user_goal,
            "optimized_prompt": optimized_prompt,
            "execution_result": execution_result
        }
```

4. Simplicity, Governance and the Future

Simplicity: Users state a high-level goal, not technical details. Our system handles the complex prompt engineering, making powerful data processing incredibly simple to use.

Governance: The centralized pipeline repository ensures standards and security. Generated prompts prevent misuse and control LLM costs, providing a clear audit trail for every run.

Cost Control: Optimized, machine-generated prompts are highly efficient. This reduces token usage, controls operational costs, and makes LLM expenses predictable and manageable for the business.

Future Scalability: Scaling is simple: add a new Beam YAML file and a prompt template to the repository. No complex code changes are needed to add new capabilities.

Tomi Ajakaiye

QUESTIONS?

tomi@bluepodconsulting.com

https://www.linkedin.com/in/a-beryl/

BEAM SUMMIT NYC 2025