Google Summer of Code 2026 Proposal

A Learning Path to Using Hardware Accelerators with **Apache Beam**

From Zero to TPU: Progressive Examples for ML Training & Inference

Applicant	Elia Liu
Email	elialiulzy@gmail.com
GitHub	github.com/Eliaaazzz
Organization	The Apache Software Foundation (Apache Beam)
Mentors	Pablo Estrada (pabloem@apache.org)
Project Size	~350 hours (Large) · Difficulty: Major

Table of Contents

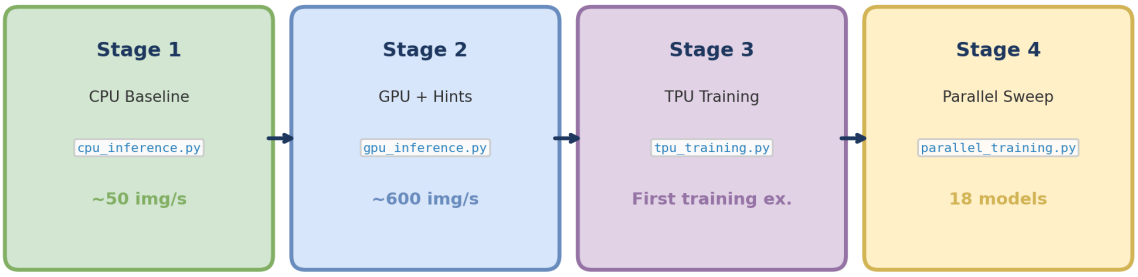
1. Abstract

Apache Beam supports GPU and TPU accelerators via Dataflow, but users face a “cliff” between a local Python script and a working accelerated pipeline. Existing notebooks jump directly to complex configurations with no intermediate steps, and **zero examples demonstrate model training** (only inference). Meanwhile, the examples that do exist reference outdated versions (`apache-beam[gcp]==2.44.0`) and break silently over time.

This project builds a **4-stage progressive learning path** — from CPU baseline to TPU-accelerated parallel training — plus a sustainable CI strategy and a reference blog post. Three technical innovations:

- **Modern `worker_accelerator` + `resource_hints` API** as the canonical way to provision accelerators
- **Shared code structure** — TPU training via `torch_xla`, GPU via native CUDA PyTorch, with a thin `get_device()` abstraction that keeps the pipeline logic identical
- **Scheduled smoke runs + nightly mocks** that validate freshness continuously without manual intervention

Figure 1. Progressive Learning Path — each stage adds ONE new concept



Users can stop at any stage and still have a working, runnable example.

2. Current Gaps

Gap #1: The Configuration Cliff

The simplest existing GPU example (`run_inference_tensorflow.ipynb`) immediately requires a custom Docker image, Runner v2, `worker_accelerator` flags, and GCS bucket setup. There is no “Hello GPU” that takes 5 minutes.

Gap #2: No Performance Comparison

No existing example benchmarks CPU vs GPU vs TPU on the same task. Users cannot answer: “Is the $3.80/hr GPU worker worth it compared to 8 CPU workers at $0.07/hr each?”

Gap #3: Inference Only, No Training

The Beam ML docs list “Model training” as a supported lifecycle step, but link to **zero examples**. All 15+ notebooks use `RunInference` for prediction. Users who need to fine-tune models inside a Beam pipeline have no reference architecture.

Gap #4: Example Rot

The TensorRT Dockerfile still references `apache-beam[gcp]==2.44.0` and `torch==1.13.1`. Without CI to validate them, examples silently decay. This project addresses freshness with **scheduled Dataflow smoke runs** — not just maintainer-triggered tests.

3. Project Goals and Deliverables

3.1 Learning Path Overview

Each stage introduces exactly one new concept. A user can stop at any stage and still have a working, useful example.

Stage	New Concept	Script	Outcome
1	CPU baseline	`cpu_inference.py`	Performance baseline
2	GPU + worker_accelerator	`gpu_inference.py`	~10–20× speedup
3	TPU + training	`tpu_training.py`	First Beam training example
4	Parallel sweep	`parallel_training.py`	18 models trained in parallel

3.2 Deliverables with Acceptance Criteria

Completion is defined at two levels. **Implementation-complete** means code reviewed, tests green, and PR approved—this is fully within my control. **Merged** means landed in trunk, which additionally depends on community review bandwidth.

D1 Four Progressive Example Scripts

- Each script is self-contained, runnable, and benchmarked with inline comments
- `worker_accelerator` API used as canonical accelerator provisioning

D2 Dockerfile + Container Configs

- `Dockerfile.gpu` (CUDA + PyTorch) and `Dockerfile.tpu` (torch_xla + libtpu)

D3 Continuous Freshness Pipeline

- Tier 1: Nightly `pytest` + `TestPipeline` on CPU (GitHub Actions, free)
- Tier 2: **Weekly scheduled Dataflow smoke run** (lightweight, ~$0.50/run) validating GPU path end-to-end

D4 Blog Post (~3000 words)

- Published on beam.apache.org with benchmark charts, architecture diagrams, and cost analysis

D5 Cost vs. Speed Cheat Sheet

- CPU / T4 / L4 / TPU comparison table embedded in blog and README

4. Technical Approach

Note on code snippets: All code blocks in this proposal are *illustrative*. Final, fully runnable examples—with complete imports, pipeline options, and error handling—will live in `examples/...` alongside the shipped SDK.

4.1 Stage 1 – CPU Baseline

Goal

Establish a performance baseline using a standard Beam `RunInference` pipeline on CPU. Runs locally in under 5 minutes with no cloud setup.

Model & Dataset

ResNet-18 pretrained on ImageNet, evaluated on CIFAR-10 (60K images, ~170 MB). Built into `torchvision`—no manual download required.

Pipeline Architecture

```
# Stage 1: cpu_inference.py
import apache_beam as beam
from apache_beam.ml.inference.pytorch_inference import
PytorchModelHandlerTensor

model_handler = PytorchModelHandlerTensor(
```

```
    model_class=torchvision.models.resnet18,
    model_params={"weights": "IMAGENET1K_V1"},
)

with beam.Pipeline() as p:
    _ = (
        p
        | "LoadCIFAR" >> beam.Create(load_cifar10_tensors())
        | "Inference" >> RunInference(model_handler)
        | "Accuracy"  >> beam.CombineGlobally(AccuracyFn())
        | "Log"       >> beam.Map(print)
    )
```

Expected output: ~50 images/sec on a modern laptop.

4.2 Stage 2 – GPU Acceleration

New Concept: worker_accelerator + resource_hints

This stage introduces the `worker_accelerator` flag and `resource_hints` API for provisioning GPU workers on Dataflow. Per the Dataflow GPU support docs, provisioning uses the `type:ACCELERATOR;count:N;install-nvidia-driver` syntax.

Note: On Dataflow, `--worker_accelerator` provisions the actual GPU/TPU workers. `resource_hints` annotates the specific transform that benefits from an accelerator (readability today, and runner placement/portability over time). In these examples, GPUs use both; TPUs are provisioned via `--worker_accelerator` only.

Pipeline Changes (Diff from Stage 1)

```
# Stage 2: gpu_inference.py  (changes from Stage 1 highlighted)
# NEW: resource hints for GPU provisioning
| "Inference" >> RunInference(model_handler).with_resource_hints(
    accelerator="type:nvidia-tesla-t4;count:1;install-nvidia-driver"
  )

# NEW: Dataflow runner options
pipeline_options = PipelineOptions([
    "--runner=DataflowRunner",
    "--project=PROJECT_ID",
    "--region=us-central1",
    "--temp_location=gs://BUCKET/tmp",
    "--sdk_container_image=CUSTOM_IMAGE",
    # For GPU workers:

"--worker_accelerator=type:nvidia-tesla-t4;count:1;install-nvidia-driver",
])
```

Custom Docker Image

```
# Dockerfile.gpu
FROM apache/beam_python3.11_sdk:2.63.0
```

```
RUN pip install torch==2.5.0 torchvision==0.20.0 \
    --index-url https://download.pytorch.org/whl/cu121
COPY stages/ /app/stages/
```

Expected output: ~500–1000 images/sec. ~10–20× speedup over CPU baseline.

4.3 Stage 3 – TPU Training

New Concept: Training on Accelerators

This is the **first Beam example that demonstrates model training** (not just inference). TPU training uses `torch_xla`; GPU training uses native CUDA PyTorch. The pipeline logic (`TrainModelDoFn`) is shared across backends via a thin `get_device()` abstraction.

Why Training in Beam?

Beam is **not** a replacement for Ray Train or PyTorch DDP for large-scale distributed pretraining. This project positions Beam for three scenarios where its pipeline model is uniquely suited: **(1)** partitioned fine-tuning (train separate models per customer/region), **(2)** hyperparameter sweeps (embarrassingly parallel grid search), and **(3)** online learning as part of a streaming pipeline.

Device Abstraction

Training uses `torch_xla` for TPU; GPU uses standard CUDA PyTorch. The `get_device()` helper selects the backend based on `PJRT_DEVICE` (set in our TPU Docker image). On GPU workers, `PJRT_DEVICE` is absent, so torch_xla is **never imported**—eliminating the risk of accidentally running XLA on CUDA hardware.

```
# device_utils.py
import os
import torch

def get_device(backend: str = "auto") -> torch.device:
    """
    Select device backend: 'tpu', 'gpu', 'cpu', or 'auto'.

    - 'tpu': use torch_xla (expects TPU runtime / PJRT).
    - 'gpu': use native CUDA PyTorch.
    - 'cpu': always CPU.
    - 'auto': use TPU only when PJRT_DEVICE=TPU (Dataflow TPU
              image), otherwise use CUDA if available, else CPU.

    Note: We intentionally do NOT attempt to run XLA on GPU
    workers.
    """
    backend = (backend or "auto").lower()

    # TPU path (explicit or environment-indicated)
    if backend == "tpu" or (
        backend == "auto"
```

```
        and os.getenv("PJRT_DEVICE", "").upper() == "TPU"
    ):
        import torch_xla.core.xla_model as xm
        return xm.xla_device()

    # GPU path (explicit or auto)
    if backend in ("gpu", "auto"):
        if torch.cuda.is_available():
            return torch.device("cuda")
        if backend == "gpu":
            raise RuntimeError(
                "GPU requested but CUDA is not available."
            )

    return torch.device("cpu")


def sync_device(device: torch.device):
    """Flush XLA computation graph. No-op for CUDA/CPU."""
    if "xla" in str(device):
        import torch_xla.core.xla_model as xm
        xm.mark_step()
```

TrainModelDoFn

```
class TrainModelDoFn(beam.DoFn):
    """Trains a model on a single accelerator per worker."""

    def setup(self):
        # BEAM_ACCELERATOR_BACKEND can be set in Dockerfile:
        #   ENV BEAM_ACCELERATOR_BACKEND=tpu   (TPU image)
        #   not set / "auto"                   (GPU image)
        self.device = get_device(
            os.getenv("BEAM_ACCELERATOR_BACKEND", "auto")
        )

    def process(self, config: TrainingConfig):
        model = build_model(config.model_name).to(self.device)
        optimizer = torch.optim.Adam(
            model.parameters(), lr=config.learning_rate
        )
        train_loader = get_cifar10_loader(config.batch_size)
        test_loader = get_cifar10_loader(
            config.batch_size, split="test"
        )

        checkpoint_path = None
        for epoch in range(config.epochs):
            for batch_x, batch_y in train_loader:
                batch_x = batch_x.to(self.device)
                batch_y = batch_y.to(self.device)
                loss = F.cross_entropy(model(batch_x), batch_y)
                loss.backward()
                optimizer.step()
                optimizer.zero_grad()
                sync_device(self.device)  # flush XLA graph
```

```
            # Checkpoint every epoch to GCS (fault tolerance)
            checkpoint_path = save_checkpoint(
                model, config, epoch,
                f"gs://{BUCKET}/checkpoints/"
            )

        accuracy = evaluate(model, test_loader, self.device)
        yield TrainingResult(
            experiment_id=config.experiment_id,
            accuracy=accuracy, model_path=checkpoint_path,
        )

    def teardown(self):
        if "xla" in str(self.device):
            import torch_xla.core.xla_model as xm
            xm.mark_step()  # flush any pending XLA ops
        elif torch.cuda.is_available():
            torch.cuda.empty_cache()
```

TPU Docker Image & Provisioning

```
# Dockerfile.tpu
FROM apache/beam_python3.11_sdk:2.63.0
RUN pip install torch~=2.5.0 torchvision~=0.20.0 \
    torch_xla[tpu]~=2.5.0 \
    -f https://storage.googleapis.com/libtpu-releases/index.html
ENV PJRT_DEVICE=TPU
ENV TPU_SKIP_MDS_QUERY=1
ENV BEAM_ACCELERATOR_BACKEND=tpu
COPY stages/ /app/stages/
```

```
# TPU Dataflow launch command
# Uses worker_accelerator with type + topology syntax.
# TPU_TYPE depends on region availability (e.g. v5e / v5 lite);
# we will document the verified value with links to Dataflow
# TPU docs once quota is confirmed during community bonding.
python tpu_training.py \
    --runner=DataflowRunner \
    --project=PROJECT_ID \
    --worker_accelerator=type:TPU_TYPE;topology:1x1 \
    --sdk_container_image=TPU_IMAGE \
    --dataflow_service_options=enable_prime
```

TPU Topology Note

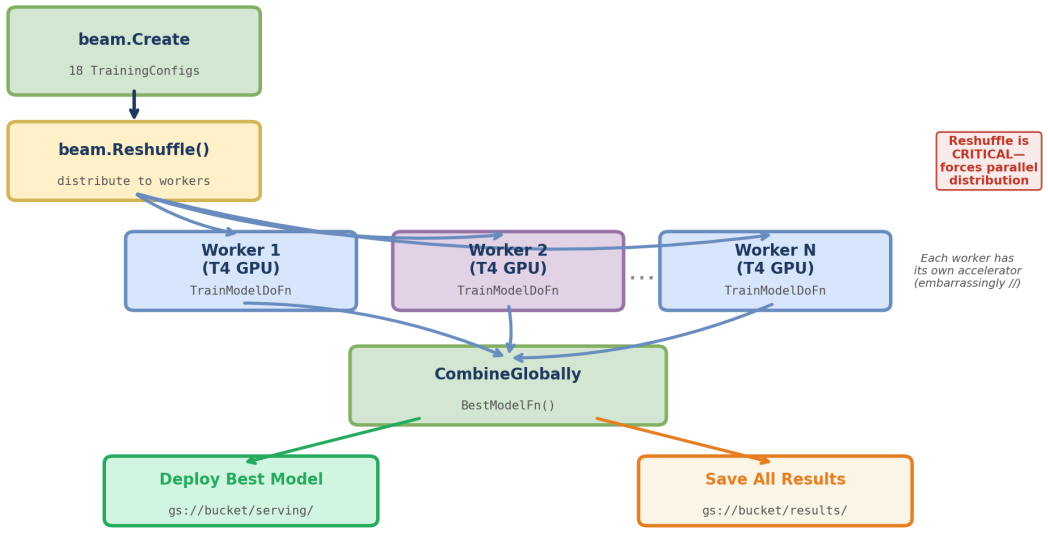
This project uses **embarrassingly parallel** training: each Dataflow worker gets one TPU chip and trains independently (`topology:1x1`). We are **not** doing multi-node distributed training. Each worker sees itself as the only TPU node. For larger topologies and multi-host training, see the Dataflow TPU documentation.

4.4 Stage 4 – Parallel Training Pipeline

New Concept: Embarrassingly Parallel ML

This stage demonstrates Beam's unique strength: distributing 18 independent training jobs across GPU/TPU workers with automatic result aggregation.

Figure 2. Stage 4 Pipeline Architecture — Parallel Hyperparameter Sweep




```python
# Stage 4: parallel_training_pipeline.py
configs = generate_grid_search(
    learning_rates=[0.01, 0.001, 0.0001],
    batch_sizes=[16, 32, 64],
    models=["resnet18", "resnet34"],
)  # 3 x 3 x 2 = 18 parallel training jobs

with beam.Pipeline(options=opts) as p:
    results = (
        p
        | beam.Create(configs)
        | beam.Reshuffle()          # CRITICAL: distribute across workers
        | beam.ParDo(TrainModelDoFn()).with_resource_hints(
            accelerator="type:nvidia-tesla-t4;count:1;install-nvidia-driver"
          )
    )
    # Select best model
    _ = (
        results
        | beam.CombineGlobally(BestModelFn())
        | beam.ParDo(DeployBestModelFn("gs://BUCKET/serving/"))
    )
    # Save all results
    _ = (
        results
        | beam.Map(lambda r: json.dumps(asdict(r)))
        | beam.io.WriteToText("gs://BUCKET/results/all")
    )
```

`beam.Reshuffle()` is **critical**—without it, Dataflow may process all 18 configs sequentially on a single worker.

worker_accelerator API Note

Both GPU and TPU use the same `--worker_accelerator` flag syntax. GPU: `type:nvidia-tesla-t4;count:1;install-nvidia-driver`. TPU: `type:TPU_TYPE;topology:1x1` (the exact TPU type depends on region availability—e.g. v5e or v5 lite; we will document the verified value once quota is confirmed). The `resource_hints` API supports GPU (via `with_resource_hints(accelerator=...)`) but does not yet support TPU provisioning. This project documents both the flag-based and hints-based paths, noting which accelerators each currently covers.

4.5 Continuous Freshness Strategy

Example rot is the **#1 threat** to this project's long-term value. The freshness strategy has three tiers, designed so that examples stay validated **automatically**—not just when a maintainer remembers to trigger a test.

Tier 1: Nightly Mock Tests (Free)

GitHub Actions runs `pytest` + `TestPipeline` on CPU every night. Validates pipeline DAG construction, DoFn lifecycle, and data flow without any cloud resources.

```
# ci/test_cpu_mock.py
def test_train_dofn_produces_result():
    """TrainModelDoFn completes 1 epoch on CPU with 10 images."""
    config = TrainingConfig(
        experiment_id="test", epochs=1,
        batch_size=2, learning_rate=0.001,
    )
    with TestPipeline() as p:
        results = (
            p
            | beam.Create([config])
            | beam.ParDo(TrainModelDoFn())
        )
        assert_that(
            results | beam.Map(lambda r: r.experiment_id),
            equal_to(["test"])
        )

def test_best_model_fn():
    """BestModelFn selects config with highest accuracy."""
    results = [
        TrainingResult("a", accuracy=0.85, model_path="gs://a"),
        TrainingResult("b", accuracy=0.92, model_path="gs://b"),
    ]
    with TestPipeline() as p:
        best = (
```

```
            p
            | beam.Create(results)
            | beam.CombineGlobally(BestModelFn())
        )
        assert_that(
            best | beam.Map(lambda r: r.experiment_id),
            equal_to(["b"])
        )
```

Tier 2: Weekly Scheduled Dataflow Smoke Run (~$0.50/run)

A **cron-triggered GitHub Action** launches a lightweight Dataflow job every Monday that runs Stage 2 (GPU inference on 1000 images) end-to-end. This catches Docker image breakage, SDK version drift, and Dataflow API changes **before** users hit them. Estimated cost: ~$0.50/week using a single preemptible T4 worker for ~3 minutes.

```
# ci/scheduled_smoke.yml
name: Weekly Dataflow Smoke
on:
  schedule: [{cron: '0 8 * * 1'}]  # Every Monday 8am UTC
jobs:
  smoke-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: google-github-actions/auth@v2
        with:
          credentials_json: ${{ secrets.GCP_SA_KEY }}
      - run: |
          python stages/gpu_inference.py \
            --runner=DataflowRunner \
            --num_images=1000 \

--worker_accelerator=type:nvidia-tesla-t4;count:1;install-nvidia-driver \
            --max_num_workers=1
      - run: python ci/verify_output.py  # check results exist
```

Tier 3: Full Integration (On-Demand)

For TPU and Stage 4 validation, `run_integration.sh` can be triggered manually via a `ci/run-integration` GitHub label. This is for pre-release validation, not continuous freshness.

Clarification on "continuous freshness": Tests run at three cadences: **nightly** (mock/DirectRunner), **weekly** (Dataflow smoke with CPU/GPU), and **on-demand** (TPU integration, triggered manually or before releases). "Continuous freshness" refers to the nightly + weekly tiers; TPU runs are intentionally on-demand to manage cost.

4.6 Cost vs. Speed Cheat Sheet

Preliminary estimates for CIFAR-10 (ResNet-18, 10 epochs). TPU values are estimates to be validated during Week 8–9.

Metric	CPU (n1-std-4)	T4 GPU	L4 GPU	TPU v5e*
Inference (img/s)	~50	~600	~900	~1200*
Train time (10 ep)	~45 min	~8 min	~5 min	~4 min*
$/hr	$0.19	$0.95	$1.22	$1.20*
Total cost (1 job)	$0.14	$0.13	$0.10	$0.08*
Speedup vs CPU	1×	~5.6×	~9×	~11×*

** TPU values are estimates; will be updated after Week 8–9 benchmarks.*

5. Timeline (12 Weeks)

Coding period: **June 2 – August 25, 2026** (aligned with the official GSoC 2026 calendar). Each week targets a single objective. If Phase 2 runs over, stretch goal time is compressed—core development time is protected.

Week	Objective	Deliverable	Acceptance Criteria
W1 (Jun 2)	Stage 1: CPU baseline + benchmarks	cpu_inference.py	Runs locally, outputs img/s
W2 (Jun 9)	Stage 2: GPU Dockerfile + worker_accelerator	Dockerfile.gpu	Docker build succeeds; GPU detected
W3 (Jun 16)	Stage 2: Deploy GPU inference to Dataflow	gpu_inference.py	10×+ speedup on Dataflow
W4 (Jun 23)	Stage 3: Design TrainModelDoFn + device_utils	Design doc	Mentor review + sign-off
W5 (Jun 30)	Stage 3: Implement training loop + checkpointing	tpu_training.py (CPU)	1-epoch train completes on CPU mock
W6 (Jul 7)	Stage 3: TPU Dockerfile + Colab TPU VM test	Dockerfile.tpu	Training runs on Colab TPU VM
★ MIDTERM(Jul 14–18)	**Stages 1–3 implementation-complete. Submit midterm evaluation.**	PRs open for review	All mock tests green
W7 (Jul 21)	Stage 3: Deploy training to Dataflow TPU	TPU Dataflow job	Accuracy ≥ 80%; checkpoint in GCS
W8 (Jul 28)	Stage 4: Parallel pipeline + Reshuffle	parallel_training.py	18 configs distributed

W9 (Aug 4)	Stage 4: BestModelFn + end-to-end sweep	Results JSON	Best model auto-selected
W10 (Aug 11)	CI: Tier 1 mocks + Tier 2 scheduled smoke	ci/ directory	Nightly + weekly smoke green
W11 (Aug 18)	Blog post + Cost vs Speed cheat sheet	Blog draft	~3000 words; benchmarks included
W12 (Aug 25)	Review feedback + docs polish + final eval	All PRs updated	CI green ≥ 14 days; eval submitted

TPU Fallback: If TPU quota is denied, Stage 3 runs on GPU (same code, different device via `get_device()`). TPU validation deferred to Tier 3 CI.

6. Testing Strategy

6.1 Unit Tests (pytest, CPU)

- `TrainModelDoFn`: verify 1-epoch training completes, produces `TrainingResult`
- `BestModelFn`: verify selection of highest-accuracy result
- `get_device()`: verify fallback chain (TPU → GPU → CPU) using mock imports
- Pickle round-trip for `TrainingConfig` and `TrainingResult` dataclasses

6.2 Scheduled Smoke Tests (Dataflow)

- Weekly GPU inference end-to-end on Dataflow (Tier 2, automated)
- Validates Docker image, SDK version, and Dataflow API compatibility

6.3 Integration Tests (On-Demand)

- Stage 3: TPU training with checkpoint written to GCS
- Stage 4: 18-config parallel sweep completes, best model selected

6.4 Container Validation

Container-first development: test locally with `docker run --gpus all` before deploying to Dataflow, ensuring custom image and dependencies are validated before cloud execution.

7. Risks & Mitigations

Risk	Detail	Mitigation

TPU quota denied	v5e quota is limited and may require manual approval.	Request during bonding period. Develop on Colab TPU VM (free). GPU fallback: same code, different device.
torch_xla + Beam conflict	torch_xla multi-processing may conflict with Beam worker harness.	Set number_of_worker_harness_threads=1. Known to work per Dataflow TPU quickstart.
Works locally, fails on Dataflow	Docker image differences, missing deps, SDK version mismatch.	Container-first dev: always test in Docker locally before cloud deploy.
Training too slow	CIFAR-10 is small, but Docker cold start + GCS data loading adds overhead.	Pre-stage data to GCS. ~2 min/epoch on T4. Pattern demo, not prod-scale.
Worker preemption	Dataflow workers can be preempted mid-training.	Checkpoint every epoch to GCS. Demonstrates fault-tolerant training.
Scope creep	Temptation to add TensorRT, streaming, multi-node distributed.	All explicitly out of scope. Single-chip embarrassingly parallel only.

8. About Me

I am Elia Liu, a final-year computer science student at the University of Melbourne. I have been contributing to Apache Beam's Python SDK and am drawn to this project because making hardware accelerators accessible to ML practitioners is one of the most impactful things the Beam community can do right now.

8.1 Prior Contributions to Apache Beam

PR #37299 Fixed a production stability bug in `ExternalTransform.expand()` where direct access to `_type_hints` raised an `AttributeError`. Replaced with `get_type_hints()`. Gave me hands-on experience with Beam's transform expansion internals.

PR #37428 Added content-aware dynamic batching to RunInference via `element_size_fn`. Enables batching by actual content cost (token count, pixel count) for efficient GPU utilization. Sits squarely in Beam ML space.

8.2 Why Me

- **Python + PyTorch:** Both merged PRs involved production-quality Python with pytest coverage. Extensive PyTorch experience from coursework and personal projects.
- **Full-stack background:** Java/React experience gives me an architectural understanding of Beam's SDK internals and runner model, useful for debugging cross-language and pipeline construction issues.
- **AI Infrastructure:** Hands-on Docker + GCP experience. Familiar with custom container workflows, Dataflow deployment, and GCS integration—exactly the toolchain this project demands.
- **Documentation focus:** I understand where jargon confuses beginners—exactly what this project needs. My PRs include detailed descriptions and inline comments.

8.3 Availability

I can dedicate approximately 30 hours/week, consistent with the 350-hour budget over 12 weeks. No major conflicts.

9. Communication Plan

Weekly video call with Pablo Estrada. Async updates on Beam dev mailing list. Draft PRs opened early for community visibility. Design docs shared as Google Docs before implementation. Review feedback addressed within 24 hours.

Pre-GSoC plan: Submit at least 1 additional PR to apache/beam before coding starts (e.g., fix a broken ML notebook).

10. References

[1] Beam ML Dataflow TPU Examples
[2] Beam RunInference API
[3] Dataflow GPU Support
[4] Dataflow TPU Support
[5] torch_xla Documentation
[6] Beam resource_hints API
[7] PR #37299 – ExternalTransform fix
[8] PR #37428 – RunInference dynamic batching