Auxia + True Theta Proposal for the Data Product “Prism”

Author: DJ Rich
Collaborators: Sumeet Kumar
Started: Jan 21, 2026
Updated: Jan 26, 2026

This document is to propose work that is high impact for Auxia and well-scoped for the True Theta Team. This proposal followed a brainstorming session with Sumeet and after feedback was provided by Max.

True Theta Team

As of writing, we have the following from True Theta for execution over a ~12-16 week period:

1. **DJ Rich**, serving as data scientist, MLE and managerial role over the True Theta team.
2. **Alex Hasha**, who has long MLE / applied experience at Capital One. He will do some architectural advisory work and some IC work. I also believe he’ll be useful outside of the specific product below.
3. **Tim Farkas, A Data Engineer / Data Scientist IC.** Former Academic / PhD in ecology with an MS in Statistics who switched to data science / data engineering 10+ years ago. Strong experience with GCP / query-munging and owning data products (resume incoming).
4. **(Optional) A data analyst.** If we think there’s a need for more data-digging resources, I have one of two analysts I can bring in. As of writing, I don’t think this is necessary.

We will also need technical advisory from Max, since he’s built much of the relevant existing code.

The Situation

Auxia has a huge number of features available for users and advertisements, but using, iterating on, and expanding the feature set has considerable friction. For a one customer, it involves editing function bodies, which creates a mental burden to understand the code context, and tracing feature metadata between several spots (e.g. ml_features, gRPC calls). Further, our choices to handle scaled data come with obfuscation; it’s not easy to visualize training data when it’s provided with streaming utilities. Lastly, our handling of this is not a unified technology. If we do something useful for one customer in this space, it does not get generalized.

The Data Product: “Prism”

I propose we develop a data product that works across customers and gives us an intuitive, clean, and well documented API. A user will specify the customer they want features / targets for, groups of features they want (the naming of which does not depend on the customer), and other parameters and will be returned a data *reference* (something that points to something to download) and compiled processor functions (which were used to generate the data and will be used in live serving, guaranteeing some protection against training / serving skew).

Python

```python
from auxia.prediction.colab import prism

# The client manages the backnd orchestration and BigQuery provisioning logic.
client = prism.Prism(project_id="auxia-gcp")

# Create a data_reference. This lets us kick off the data generating job, knows the state
of BigQuery job / the size of the resulting data, and can provide the preprocessing
callable that generated the features and can be used in serving.
data_reference = client.create_data_request(
    customer_id="1943",
    time_window=("2026-01-10", "2026-01-23"),
    surface='b63',
    features=[
        "user.demographics",
        "user.behavioral_28d",
        "treatment.clicks_and_views_28d"
    ],
)

print(data_reference.metadata.expected_rows) # Tells us the expected numbers of rows.

# Kicks off the job that creates the data and eventually stores it in GCS.
data_reference.run()

print(data_reference.status) # Tells us the status of the job

df_pandas = data_reference.to_pandas() # Downloads the data from GCS into a pandas format.
df_polars = data_reference.to_polars() # Downloads the data from GCS into a polars format.
dict_tf = data_reference.to_tf_datasets() # Downloads the data from GCS into a dictionary
mapping from feature names to tf tensors.

# For large scale datasets, use the streamer
streamer = data_reference.get_streamer()
for batch in streamer.stream(batch_size=50000):
    model.partial_fit(batch)

# This provides the exact transformation logic used for training, ensuring parity with the
live serving environment. This should get packaged with the deployment.
processor = data_reference.get_processor()
```

The advantages of this approach are:

- **Standardized API**: This standardizes the request and reception of data. What we build here will work for all customers, by design.
- **Format Flexibility**: Support for both Pandas and Polars allows teams to choose the most efficient memory representation for their specific task, while the Streaming API provides a safety valve for massive datasets that exceed local RAM.
- **Unified Metadata**: Prism centralizes the mapping between internal `data_field_ids` and human-readable feature names, removing the need for engineers to cross-reference multiple documentation sources or gRPC definitions.
- **OOM Prevention**: By returning a `data_reference` first, engineers can inspect the query's scale (expected rows, byte size) before pulling results into memory. This allows for informed decisions between a direct `to_pandas` call or using the streamer for larger-than-RAM datasets.
- **Parallelism**: Because the `create_data_request` call is non-blocking and returns a reference immediately, an engineer can kick off dozens of training jobs in parallel across different customers or time windows, then poll the references for completion.
- **Unified Logic**: Prism provides a single source of truth for complex data operations. For example, the specific logic used to join user features with treatment features is encapsulated within Prism, ensuring it remains consistent across all customers regardless of their underlying schema differences.
- **Training / Serving Skew Protection**: The `get_processor` method returns the exact transformation logic used to generate the data. By deploying this same "processor" in the live API, we guarantee that features are computed identically at both training and inference time.

Build Strategy / Execution

Phase 1: Develop the Interface, Testing and System's Design (~2 weeks)

The first priority is defining the user inference with Prism. We need to separate how a data scientist thinks about features from how Prism retrieves them. This should involve writing desired API calls (a bit like what is shown above, but many more examples), getting feedback on them, and converging on a design. Also, we should write the desired *results* as well (in a

variety of formats, polars / tf.Dataset). This would set us up with test driven development and force us to develop a horizontal technology from the start.

This will also motivate the system's design discussion. The API and tests will make certain requirements clear, and this will motivate discussion of architectural questions. Some examples:

- How should the semantic mapping layer (mapping between customer-agnostic feature groups to specific `data_field_id`s) be designed and updated over time?

- How do we handle training / serving skew with the returned TF-compiled `processor` function? Should we centralize data operations in tensorflow and convert to polars or pandas after the fact to ensure equivalence? Or should our centralized data object be provided by pyarrow?

- How do we enforce the SLAs of upstream sources, like `ml-features`? Prior to kicking off large data-producing jobs, we should ensure the upstream data is in the state it needs to be.

- How will we handle cost monitoring? Should we use the `CostMonitor`?

- Prism needs to standardize handling of the below items. This handling will actually need to be parameterized (e.g. with lookback-window length for joining) in a way that balances accuracy and speed.
    - Point-in-time join semantics to prevent future leakage.
    - Handling of missing feature rows (`INNER` vs `LEFT` join decision)
    - Surface filtering (`WHERE` clause on surface_id)
    - Date range filtering (start_date, end_date parameters)

- How do we handle categorical variables? We certainly do not want to return a categorical column with strings as values, because this will inevitably get processed into some numeric format prior to being fed to the model. So Prism should do the processing, so it can ensure it's the same across training / serving environments. For example the vocabulary will need to be saved / reused across environments.

Phase 2: Build / Execute (~2 - 6 weeks)

With the architectural choices agreed upon, we need to build those components. With Claude Code, this should not take especially long. I leave 2 - 6 weeks because it highly depends on what we decide to build in the previous stage and because as we build, we may need to rethink architectural choices (a common experience).

Development operations will be:

- **Component-wise PRs with reviews by Max / Sumeet with unit tests.** PRs should be small and contribute a component that does one job well. For example, provide the query-generating function or class that standardized the pulling of training data. Also, we will use unit tests to demonstrate the function of the component and to develop defensively.

- **Integration tests and demonstrative notebooks.** After the components are built, we will finalize the development of Prism by running integration tests and writing notebooks that demonstrate how it's used.

Phase 3: Migration and Documentation (~2 - 6 weeks)

This phase will be involved in migrating existing model pipelines onto the use of Prism. The purpose is to reduce tech debt by removing the need to maintain and update old data-generating code. Doing this will involve:

1. **Setting up testing to make sure pre-and-post migrated code produces identical models.**

2. **Migrate data-generating code** (in Colab notebooks and Auxia's source code) onto Prism and iterate on tests until they are satisfactory.

3. **Develop finalized documentation and hand-off**, demonstrating how to leverage the unified API for cross-customer feature pulling and general diagnostics.

Drawbacks, Risks, and Challenges

To ensure the project is successful, we should anticipate the following:

- **Bespoke Logic**: The current code has customization, like hardcoded lists for specific surfaces or custom date-part logic. Converting these bespoke hacks into a standardized API might result in a "lowest common denominator" tool that doesn't quite meet the needs of more complex, one-off customer requests. We will have to be careful designing the data engineering components that can really be centralized.

- **Multi-Format Maintenance**: Supporting equivalent outputs across Pandas, Polars, and Streaming formats introduces a significant testing burden to ensure "parity of truth." Subtle differences in how these libraries handle things like null types or integer overflows could lead to inconsistent model behavior depending on the format chosen.

- **Migration Friction**: Transitioning existing pipelines requires an upfront investment to replace deeply embedded, hardcoded logic with Prism API calls. This "migration tax" may create a temporary period of technical debt where developers must maintain two

different data-fetching paradigms simultaneously.

- **Scaling Abstraction and Guardrails**: Prism cannot fully hide the reality of massive datasets; users will still need to choose between `to_pandas` and `get_streamer` based on the specific customer's data volume. Prism can provide guardrails to guide the user, but it’s unlikely we can fully anticipate scaling complexities, especially since our customers vary widely in their scale.