Starlet: An End-to-End Python Library for Scalable Geospatial Tiling and Vector-Tile Serving

Tarlan Bahadori
tbaha001@ucr.edu
UC Riverside
California, USA

Rohan Bennur
rbenn020@ucr.edu
UC Riverside
California, USA

Shaolin Xie
shaolinx@usc.edu
University of Southern California
California, USA

Ibrahim Sabek
sabek@usc.edu
University of Southern California
California, USA

Ahmed Eldawy
eldawy@ucr.edu
UC Riverside
California, USA

Abstract

Interactive exploration of large geospatial datasets requires a practical path from raw vector data to low-latency web map tiles. This paper presents Starlet, a pip-installable Python library for end-to-end geospatial tiling and vector-tile serving over GeoParquet. Starlet ingests GeoParquet or GeoJSON, partitions the data into balanced spatial shards, builds a Web Mercator density grid for fast tile-occupancy tests, and generates Mapbox Vector Tiles either eagerly into an on-disk pyramid or lazily on first request. Its serving layer combines memory caching, disk-backed tiles, and on-demand generation, while its streaming writer bounds open files and per-round processing during ingestion. On commodity laptop hardware, Starlet partitions and indexes a 17.6 GB OpenStreetMap extract with 43 M features containing one billion vertices in 38.5 minutes, while keeping peak RSS below 9.6 GB. These results show that a lightweight Python-native stack can provide scalable ingestion, compact tile generation, and interactive vector-tile serving for large geospatial datasets.

Keywords

GeoParquet, Vector Tiles, Interactive Map Visualization

1 Introduction

Web-scale geospatial visualization typically requires three stages: transforming source data into a tile-friendly storage layout, constructing or selecting a tile pyramid, and serving tiles to an interactive client renderer. Existing systems address different parts of this pipeline, but few provide an end-to-end solution that is both native to modern columnar spatial data and easy to embed in Python workflows. Some tools convert source files into static tile archives but are primarily designed for single-machine batch conversion [3, 13, 16, 19]. Other systems provide dynamic vector-tile serving, but commonly assume a relational spatial backend or an externally managed tile archive [5, 10, 11, 15]. Classical geospatial servers expose standard map services over file- or database-backed stores, but generally rely on separately maintained pyramids or server-side rendering pipelines. Distributed spatial processing frameworks, meanwhile, provide scalable partitioning and analysis but typically require additional tooling to convert their outputs into web-client-compatible vector tile [6, 7, 14, 21].

GeoParquet [18] provides a portable columnar representation for spatial data. Together with the Mapbox Vector Tile specification [12], this creates an opportunity to separate storage, indexing, tile generation, and presentation in a clean and interoperable way. However, this separation is rarely available in a single Python distribution that ingests GeoParquet or GeoJSON, exposes tiling APIs, and serves tiles directly to standard web-map clients.

Starlet[1] addresses this gap with an end-to-end Python library for scalable geospatial tiling and vector-tile serving. As shown in Figure 1, Starlet connects data preparation, MVT generation, tile serving, and prompt-based styling in a single workflow. Starlet partitions GeoParquet or GeoJSON inputs into a tile-friendly GeoParquet shard layout, builds a Web-Mercator vertex-count density grid, generates Mapbox-compliant vector tiles either eagerly into an on-disk pyramid or lazily on first request, and serves tiles, dataset metadata, attribute statistics, and bounded feature downloads through a lightweight HTTP interface. Its design is organized around four components: (i) a round-based partitioning and writing pipeline that combines an R*-Grove-inspired split policy [2, 20] with optional Morton-order row sorting while bounding the number of concurrently active writers and rows processed per round; (ii) a 4096×4096 prefix-sum density grid that supports rectangular range-count queries in $O(1)$ and prunes empty or low-density tiles before geometry decoding; (iii) an MVT generation and serving layer with memory caching, disk-backed tiles, on-demand generation, and topology-preserving Douglas–Peucker simplification; and (iv) a streaming attribute-sketch module, including HyperLogLog cardinality estimates [8] and SpaceSaving Top-K summaries [17], for dataset previews and sketch-driven styling.

2 GeoParquet Partitioning and Index

The goal of this step is to spatially partition a large input dataset into nearly equi-sized shards and write each one as a GeoParquet file. This structure efficiently supports interactive map exploration by quickly running range queries on the current map view to retrieve the data records. We observe that tile-based map exploration need to handle requests for MVT map tiles which are small axis-aligned bounding boxes so the index need not support arbitrary point or k-NN queries. This shapes three design choices: (i) partition data into nearly equi-sized disjoint shards and encode the bounding box

[1] Package and installation: https://pypi.org/project/starlet/; source code: https://github.com/ucr-bdlab/starlet; project page and demo: https://starmap.cs.ucr.edu.

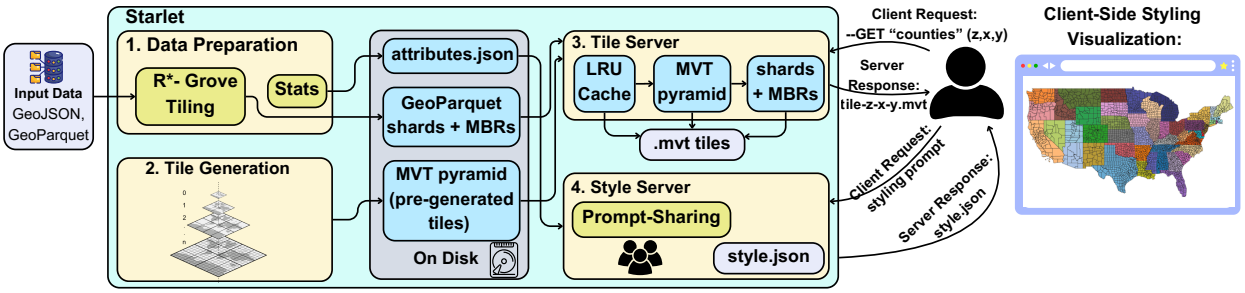


Figure 1: Starlet overview. Input GeoJSON or GeoParquet is partitioned into indexed GeoParquet shards, summarized with a prefix-sum density grid and attribute statistics, and used to generate an optional disk MVT pyramid. At serving time, tile requests are answered from the LRU cache, pre-generated MVTs, or on-demand generation from indexed shards; the same statistics also support client-side styling through generated `style.json` rules.

of each shared in its filename; (ii) sort geometries inside each shard according to a space-filling curve as a light-weight local spatial index; (iii) maintain a size-based histogram to quickly avoid empty requests without opening any shard. Unlike discrete global grids such as H3 [4] or S2 that fix a global cell hierarchy, Starlet employs R*-Grove that adapts to data density and are recovered at shard-open time from the filename, with no auxiliary index.

Sample-driven R-Grove split.* We follow R*-Grove [20]: produce a small, balanced set of (near-)disjoint partition MBRs by recursively splitting a centroid sample, using the R*-tree margin/overlap/area triplet [2] as the split score. The sample is drawn in one streaming pass via reservoir sampling; partitions are sized either uniformly or histogram-weighted so dense regions get more, smaller partitions.

Streaming centroid assignment. Starlet assigns rows to partitions in a streaming plane-sweep pass. For each batch, geometries are ordered by centroid x, an active window of candidate partitions is maintained by $x_{\min}$, and only active MBRs are tested for containment. Centroids inside a partition are assigned to that partition; those outside all active MBRs fall back to the partition requiring the smallest MBR expansion, while degenerate geometries use the smallest-area partition. Each output filename encodes its partition MBR, allowing the serving layer to recover the shard map at startup without a separate index.

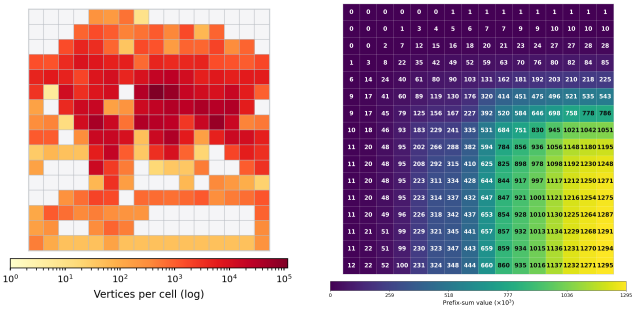
Round-buffered parallel writer. A naive "one writer per partition" policy exhausts file descriptors and DRAM at scale. Starlet's `WriterPool` buffers Arrow tables by tile ID in memory; the orchestrator emits partitions in *rounds*, admitting at most K active tile writers plus one overflow shard per round (default K=64); within each round, `WriterPool` separately controls the number of parallel file-write tasks, defaulting to the available CPU parallelism. Rows for tiles outside the open set spill to the overflow shard and are picked up by the next round. The bound is on file descriptors and per-round row throughput, not bytes resident. At end-of-round, shards are sorted in parallel (optionally along a Z-order Morton code on centroid coordinates) and written with a file-level bbox in the GeoParquet 1.1 `geo` metadata [18] and filename, enough for the serving layer to prune whole shards by MBR intersection. Together with the fixed-size density grid, this round-based design keeps ingestion memory bounded by the round budget rather than the full input size; in our largest run, Starlet processes 17.6 GB and 42.77 M polygons with peak RSS of 9.54 GB

Density grid and streaming sketches. In parallel with shard writing, Starlet accumulates a Web-Mercator (EPSG:3857) vertex-count grid of 4096×4096 cells and persists its 2D prefix sum ; a 4-corner difference answers "how many vertices fall inside tile (z, x, y)?" in $O(1)$, exactly at $z \le z_{\text{hist}}$=12 and as a scaled estimate above. The same streaming pass populates per-column attribute sketches, min/mean/variance plus HyperLogLog [8] for numeric columns, HyperLogLog with SpaceSaving [17] Top-K for categorical/text columns, and type counts, vertex counts, and the global MBR for geometry; produced in one library-level pass. The output is plain GeoParquet 1.1, so the directory is readable by GeoPandas, DuckDB, Sedona/GeoSpark [21], and Beast [7].

3 MVT Generation and Serving

The MVT layer converts the partitioned GeoParquet store into Mapbox Vector Tiles (MVT) [12] that can be rendered by standard web clients such as MapLibre [14], Leaflet, and OpenLayers. Starlet supports two tile-generation modes. In *eager* mode, the system uses the prefix-sum density grid to identify tiles that are likely to contain data, then streams the indexed shards and assigns decoded geometries to the selected tiles. In *lazy* mode, tiles are materialized on demand: the server loads shard MBRs from filenames at startup, restricts each request to shards whose MBR intersects the tile envelope, and renders the requested tile inline. The density grid is used primarily for eager tile selection, where its prefix-sum counts identify nonempty or sufficiently dense tiles before geometry decoding.

Histogram-driven tile assignment. Eager tile generation benefits from avoiding empty or low-density regions, especially at high zoom levels where most possible tiles contain no features. For each candidate tile (z, x, y), Starlet queries the prefix-sum density grid to estimate whether the tile contains at least τ vertices before reading any geometry. Counts are exact for $z \le z_{\text{hist}} = 12$ and estimated above the native histogram resolution by distributing each histogram-cell count uniformly over its descendant tiles. To



(a) Density grid H (b) Prefix sum P

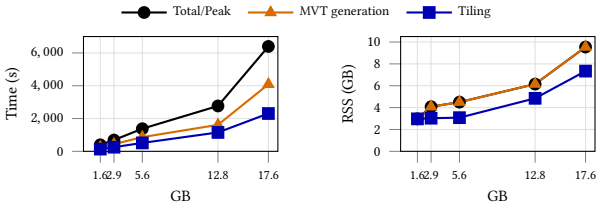
Figure 2: Density-driven tile pruning. Starlet rasterizes vertices into a fixed density grid H and stores its 2-D prefix sum P. Any tile-aligned rectangle can then be counted in $O(1)$ using four corner lookups, allowing empty regions to be skipped before eager MVT generation.

keep generated tiles small enough for interactive rendering, Starlet exposes the per-tile feature cap as a tunable parameter; in our experiments, we set this cap to 25,000 features and enforce it using a shared-priority bounded top-K sampler.

Each geometry receives a single random key when it enters the assignment pipeline, and all tiles that overlap the geometry use the same key when deciding whether to retain it. This makes keep/drop decisions consistent across neighboring tiles and reduces seam artifacts compared with independent per-tile reservoir sampling.

Figure 2 illustrates Starlet's density-driven pruning mechanism. Starlet first rasterizes geometry vertices into a fixed density grid H, where each cell stores a vertex count. It then computes the two-dimensional prefix sum P, so the total count inside any tile-aligned rectangle can be recovered in $O(1)$ using four corner lookups. Candidate tiles whose estimated counts fall below the threshold τ are skipped before geometry decoding, reducing eager tile-generation work without scanning the underlying shards. The figure shows the computation on a 16×16 grid for clarity; the implementation uses a 4096×4096 grid over the global Web Mercator extent.

Geometry streaming, clipping, and encoding. In eager (pre-generated) mode, Starlet streams over the indexed shards and assigns each geometry to its overlapping selected tiles before requests arrive. In lazy (on-demand) mode, Starlet materializes only the requested tile by filtering filename-encoded shard MBRs against the tile envelope and decoding the matching geometries inline. When enabled, Starlet can also write per-row covering boxes and bounded Parquet row groups, allowing lazy requests to use predicate pushdown and decode only intersecting row groups. In both modes, WKB geometries are decoded with Shapely 2 and reprojected from EPSG:4326 to Web Mercator using PyProj. The renderer pads each tile envelope by a fixed 256/4096-tile margin, applies topology-preserving Douglas–Peucker simplification with tolerance 0.05% of the tile width, repairs invalid geometries when necessary, maps coordinates to the MVT $[0, 4096]$ integer extent, and encodes the result with `mapbox_vector_tile`. Starlet can also package the on-disk pyramid as a single PMTiles archive.



(a) Runtime (b) Memory

Figure 3: Indexing scalability on OSM-Parks. Runtime and peak memory are shown for inputs from 1.6 to 17.6 GB, corresponding to 2.49–42.77 M polygons. Total runtime increases from 397 s to 6,397 s, while peak RSS remains below 9.6 GB.

Tile serving. At request time, `VectorTiler` checks three storage levels in order: an in-memory LRU cache, an on-disk tile pyramid at `<dataset>/mvt/z/x/y.mvt`, and the on-demand generator. Tiles generated lazily are promoted to the memory cache, so repeated requests avoid recomputation. The same Flask application also exposes dataset metadata, attribute statistics, sample records, and bounded feature downloads in GeoJSON or CSV, providing a compact HTTP interface for interactive web-map clients.

4 Evaluation

We evaluate Starlet on a commodity Apple M-series laptop with 11 cores, 18 GB DRAM, and an NVMe SSD. The evaluation uses three UCR-STAR [9] datasets: **OSM21-Parks** GeoParquet extracts of 1.6 GB up to 17.6 GB; **TIGER-Rails-2018**, a 49 MB full-US line dataset with 145,116 features; and 128 MB US **TIGER-Counties-2018** polygons dataset.

Ingestion scalability. Figure 3 reports tiling time, MVT-generation time, total wall time, and peak resident memory across OSM-Parks subsets ranging from 1.6 to 17.6 GB and 2.49 M to 42.77 M polygons containing 1 billion vertices. We used $\tau = 25\,000$ and pre-generated tiles up to zoom level 7. Total runtime grows linearly with feature count, increasing from 397 s to 6,397 s as input size grows by 17×. Peak memory increases from 2.97 to 9.54 GB over the same range: the first subset already includes fixed runtime and data-structure costs, while streaming rounds keep additional memory growth sublinear in the input size. The full dataset is processed end-to-end without running out of memory.

Serving latency under concurrent requests. Figure 4 reports tile-serving latency as the number of concurrent clients increases from 1 to 20 on Parks dataset. The experiment uses the threaded Starlet Flask server and concurrent HTTP clients requesting the same tiles over the loopback interface. We compare cached pyramid tiles with on-demand tiles generated from GeoParquet on first request. Cached requests remain low-latency: median latency increases from 0.3 ms with one client to 4.7 ms with 20 clients, with p95 below 5 ms. On-demand requests are more expensive under contention: median latency increases from 6.2 ms to 79 ms, and p95 reaches about 0.4 s at 20 clients. These results show that Starlet's cached serving path remains interactive under concurrent access, while cold on-demand generation benefits from caching after the first request.

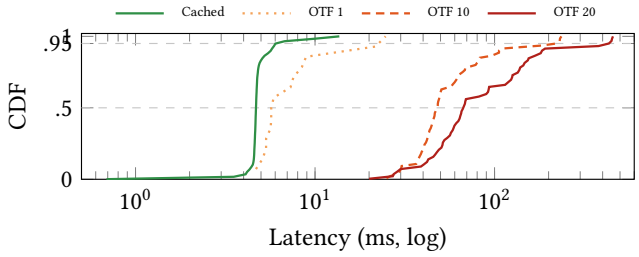


Figure 4: Serving-latency CDF under concurrent same-tile requests (log x-axis). Cached requests remain near 5 ms across loads, while cold on-demand generation shifts right and develops a longer tail as concurrency increases from 1 to 20 clients ($p95$: 22 ms to 0.43 s).

Table 1: Storage footprint of loose `.mvt` files versus a PMTiles archive for $z = 0 \ldots 7$ with 4 KB disk blocks. "On-disk" includes block allocation overhead; "Reduction" is relative to loose files on-disk usage.

Dataset	Layout	Tile bytes	On-disk	Reduction	Padding
Counties	Loose files	5.16 MB	5.59 MB	—	8%
	PMTiles	2.96 MB	2.96 MB	47%	<0.1%
Rails	Loose files	22.91 MB	23.58 MB	—	3%
	PMTiles	8.03 MB	8.03 MB	66%	<0.1%
OSM-Parks	Loose files	771.5 MB	782.5 MB	—	1%
	PMTiles	282.0 MB	282.0 MB	64%	<0.1%

Output compactness. Starlet can export the generated pyramid to a single gzip-compressed PMTiles [11] archive (`build --pmtiles`). Table 1 shows this packing cuts on-disk storage by 47–64% relative to the loose `.mvt` tiles, by eliminating per-file block overhead and storing compressed tile payloads contiguously.

5 Visualization Styling and Prompt Sharing

LLM-assisted style reuse. Starlet-generated MVTs provide the data substrate for interactive maps, but users must still specify how geometries and attributes should be visualized. In practice, this often requires framework-specific JSON or HTML describing layers, encodings, legends, labels, and interactions, which can be a barrier for users without web-mapping expertise.

To make this step more accessible, Starlet includes an optional style-reuse interface inspired by recent work on LLM-assisted geospatial styling [1]. Users can select from a repository of reusable visualization templates, each annotated with metadata such as supported geometry types, application domains, and example outputs. Templates capture common cartographic patterns, such as choropleth maps, hotspot maps, clustered point views, and transportation-network renderings. In addition to sharing final style templates, Starlet allows users to share the natural-language prompts used to create or refine them. These prompts provide a lightweight record of user intent, making it easier for others to understand, adapt, and extend a visualization design for related datasets.

Given a selected template and a target dataset, Starlet passes the template metadata and dataset summary to an integrated LLM-based styling engine. The engine produces a dataset-specific style specification in structured JSON, which is rendered as a self-contained HTML map that loads Starlet-generated MVTs in a standard web-mapping framework such as MapLibre. Users can preview the result and optionally refine it using natural-language instructions. This component illustrates how reusable templates and LLM-assisted adaptation can help users turn generated tiles into usable visualizations with less manual styling effort.

6 Conclusion

Starlet provides an end-to-end Python-native pipeline for indexing, generating, and serving vector tiles from GeoParquet and GeoJSON inputs. By combining balanced spatial partitioning, a prefix-sum density grid, streaming MVT generation, and a cache–disk–on-demand serving hierarchy, Starlet turns large vector datasets into interactive web-map tiles with modest single-machine resources. Experiments show that ingestion scales to multi-gigabyte OSM extracts, density-driven pruning skips most empty candidate tiles, cached serving remains low-latency under concurrent requests, and PMTiles packing substantially reduces on-disk footprint. Future work includes distributed ingestion and a MapLibre style generator driven by Starlet's streaming attribute summaries.

References

[1] Tarlan Bahadori et al. 2025. LASEK: LLM-Assisted Style Exploration Kit for Geospatial Data. *Proc. VLDB Endow.* 18, 12 (2025). doi:10.14778/3750601.3750690
[2] Norbert Beckmann et al. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data.* doi:10.1145/93597.98741
[3] Michael Black. 2024. Planetiler: Single-Host OSM-PBF to MBTiles/PMTiles Converter. https://github.com/onthegomap/planetiler Accessed 2026-05-12.
[4] Isaac Brodsky. 2018. H3: Uber's Hexagonal Hierarchical Spatial Index. In *Uber Engineering Blog.* https://www.uber.com/blog/h3/
[5] Development Seed. 2024. TiPg and TiMVT: PostGIS-Based Dynamic Vector Tile Servers. https://github.com/developmentseed/tipg Accessed 2026-05-12.
[6] Ahmed Eldawy et al. 2015. SpatialHadoop: A MapReduce Framework for Spatial Data. In *Proceedings of the 31st IEEE International Conference on Data Engineering.*
[7] Ahmed Eldawy et al. 2021. Beast: Scalable Exploratory Analytics on Spatio-Temporal Data. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management.* doi:10.1145/3459637.3481897
[8] Philippe Flajolet et al. 2007. HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm. In *Proceedings of the 13th Conference on Analysis of Algorithms.*
[9] Saheli Ghosh et al. 2019. UCR-STAR: The UCR Spatio-Temporal Active Repository. In *SIGSPATIAL Special*, Vol. 11. doi:10.1145/3377000.3377005
[10] Go Spatial. 2024. Tegola: A Vector Tile Server Written in Go. https://tegola.io/ Accessed 2026-05-12.
[11] Brandon Landers. 2023. PMTiles: A Single-File Archive Format for Pyramids of Tiled Data. https://protomaps.com/docs/pmtiles Accessed 2026-05-12.
[12] Mapbox. [n. d.]. Mapbox Vector Tile Specification, Version 2.1. https://github.com/mapbox/vector-tile-spec Accessed 2026-06-17.
[13] Mapbox. 2024. Tippecanoe. https://github.com/mapbox/tippecanoe Accessed 2026-05-12.
[14] MapLibre Contributors. 2024. MapLibre GL JS: Open-Source WebGL Map Renderer. https://maplibre.org/ Accessed 2026-05-12.
[15] MapLibre Contributors. 2024. Martin: Vector Tile Server from Large PostGIS Databases. https://martin.maplibre.org/ Accessed 2026-05-12.
[16] Maptiler. 2024. tileserver-gl: Vector and Raster Tile Server. https://github.com/maptiler/tileserver-gl Accessed 2026-05-12.
[17] Ahmed Metwally et al. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory.* doi:10.1007/978-3-540-30570-5_27
[18] Open Geospatial Consortium. 2024. GeoParquet Specification v1.1.0. https://geoparquet.org/releases/v1.1.0/ Accessed 2026-05-12.
[19] OpenMapTiles Contributors. 2024. OpenMapTiles: Open Vector Tile Schema and Pipeline. https://openmaptiles.org/ Accessed 2026-05-12.
[20] Tin Vu et al. 2020. R*-Grove: Balanced Spatial Partitioning for Large-Scale Datasets. In *Frontiers in Big Data*, Vol. 3. doi:10.3389/fdata.2020.00028
[21] Jia Yu et al. 2015. GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data. In *Proceedings of the 23rd ACM SIGSPATIAL.*