you don’t have to rewrite everything just because you want millions of users. Plenty of huge products run critical paths in Python (Instagram, Pinterest, parts of YouTube). What matters more than the language is **architecture, latency budgets, and how you scale**.

### **When Python is perfectly fine**

* **Backend APIs & business logic:** FastAPI/Starlette with uvicorn/gunicorn can handle very high RPS when you add instances behind a load balancer and use connection pooling.
* **ML/NLP scoring services:** Most inference stacks are Python at the edges but call **high-performance C/CUDA kernels** under the hood (PyTorch, TensorRT, ONNX Runtime).
* **Asynchronous I/O bound work:** Python’s asyncio + uvloop shines for network and disk I/O; scale horizontally with multiple workers/pods.

### **Where you may need more than Python**

* **Hot CPU bottlenecks** (heavy number crunching, image ops): move just those hotspots to **C/C++/Rust** extensions or a separate microservice in Go/Java/Rust. Keep the rest in Python.
* **Extreme tail latency guarantees** (e.g., p99 < 20 ms at very high RPS): sometimes easier with a compiled service for that specific path—but again, only the hot path, not the whole codebase.

### **Make future rewrites optional (not inevitable)**

Design now so you *can* scale later without a rewrite:

* **Clear service boundaries:** expose your logic via HTTP/gRPC with an OpenAPI contract; the client doesn’t care what language is behind it.
* **Keep domain logic pure:** isolate “CleanScore” rules in a small, well-tested module; if you ever port it, it’s a single unit.
* **Observability first:** logging, metrics, tracing (e.g., OpenTelemetry) so you know where real bottlenecks are before changing languages.

### **Proven scaling patterns (Python)**

* **Horizontal scale:** containers (Docker/K8s), autoscaling, multiple gunicorn workers per pod.
* **Caching:** Redis/Memcached for repeated computations; CDN for static assets/exports.
* **Queues:** Celery/RQ/Kafka for slow/background jobs (file parsing, batch scoring).
* **Databases:** use proper indexes and a connection pooler (e.g., PgBouncer); read replicas for heavy reads.
* **Performance tweaks:** asyncio, uvloop, avoid blocking calls, pre-load models, warm pools.

### **Mobile reality check**

* **The mobile app itself** is typically **Swift/Kotlin** or cross-platform (**React Native/Flutter**). Your Python service lives on the **server**. Millions of users hit **APIs**, not your Python interpreter directly.

### **Practical guidance**

Start with Python. Measure. If a profiler shows a true compute bottleneck, peel off *just that part* into a faster extension/service. A full-stack rewrite is a last resort; a **polyglot microservice** approach lets you scale without throwing away working code.

**Bottom line:** Python is a solid long-term choice for your core service. Good architecture + targeted optimizations beat “rewrite in X” every time.