International Journal of Advance Research in Science, Management, Engineering and Technology ISSN: 3139-1141
Volume 1, Issue 5 | November - December 2026 | www.ijarsmet.com

Java Spring Boot vs Python FastAPI for Serving ML/LLM Inference APIs: A Comparative Study of Performance, Concurrency, Ecosystem Fit and Deployment Cost

Radhika Nitinsingh Bisen

Master of Computer Applications, Zeal Institute of Business Administration, Computer Application and Research (ZIBACAR)
Savitribai Phule Pune University, Pune, India
Radhikabisen930@gmail.com

Abstract—

Machine learning models are of little use until they are wrapped in a service that other applications can call. In practice, this serving layer is built with one of two very different families of tools. Python frameworks such as FastAPI sit right next to the libraries in which most models are trained, while Java frameworks such as Spring Boot dominate enterprise back ends where teams already have mature security, monitoring and deployment practices. The question a team usually faces is whether to keep the model API in Python or to move it to the JVM stack the rest of the organisation already runs. This paper compares the two options for serving both classical ML models and large language model (LLM) workloads. The comparison covers the programming and concurrency model of each framework, how each one loads and runs models, support for token streaming, request validation, observability, container footprint, start-up behaviour and developer productivity. A reproducible benchmark design is proposed in which the same ONNX model and the same request payloads are served by a FastAPI service and a Spring Boot service, and both are measured under identical load for latency percentiles, throughput, CPU use, memory use and start-up time. For LLM workloads, the paper looks at the case where the web framework acts as a gateway in front of a dedicated inference engine and streams generated tokens back to the client. The study is intended to give practitioners a clear, evidence-based way of choosing between the two frameworks rather than defaulting to habit.

Keywords--- *Spring Boot; FastAPI; ML Model Serving; LLM Inference; REST API; Asynchronous Programming; ONNX Runtime; Latency and Throughput; Microservices*

I. INTRODUCTION

Over the last few years the work of a machine learning team has moved well beyond training. A model that scores well in a notebook still has to be exposed as an API, scaled, secured, monitored and updated without breaking the applications that depend on it. The language in which a model is trained is almost always Python, but the language in which the surrounding business system is written is very often Java. This creates a real design decision: either the model is served from a Python web framework, or it is exported to a portable format and served from the JVM.

FastAPI is a modern Python framework built on the ASGI standard, with automatic request validation through Pydantic and automatic OpenAPI documentation. Spring Boot is the most widely used Java framework for building production web services and comes with a very large ecosystem for security, data access, configuration and monitoring. Both can expose a model through a REST endpoint within a few dozen lines of code, so the difference between them only shows up under load, in operations, and in how well each one fits the team that has to maintain it. The rise of LLM-based applications has made the comparison more interesting, because LLM responses are long-running and are usually streamed token by token, which puts different demands on the web layer than a quick classification call does.

II. BACKGROUND OF THE STUDY

Model serving can be organised in several ways. In the simplest form the model is loaded into the same process as the web framework and each request calls the model directly. A second form separates the two, so that a dedicated inference server holds the model and the web service forwards requests to it. A third form uses managed platforms offered by cloud providers. The first two are the ones in which the choice of web framework matters most, and they are the focus of this paper.

FastAPI runs on Starlette and is normally deployed behind Uvicorn, sometimes with several worker processes managed by Gunicorn. Because of the Global Interpreter Lock in standard CPython, a single Python process executes Python bytecode on one thread at a time, so parallelism for CPU-bound work is usually obtained by running multiple processes. Libraries such as NumPy, PyTorch and ONNX Runtime release the lock while they run native code, which softens this limit for inference. Asynchronous endpoints are well suited to waiting on network calls, for example when a

gateway forwards a prompt to an LLM engine and relays the answer.

Spring Boot applications run on the JVM and, in the traditional servlet stack, use a pool of platform threads to handle requests. Spring WebFlux offers a non-blocking, reactive alternative, and recent Java versions add virtual threads that make blocking-style code cheaper to scale. Inference on the JVM is possible through libraries such as ONNX Runtime for Java, Deep Java Library (DJL) and TensorFlow for Java, and the Spring ecosystem now also includes Spring AI for working with LLM providers. The JVM has a longer warm-up period because of just-in-time compilation, but in return it offers mature garbage collection, strong tooling for profiling and a long track record in high-throughput services.

III. Research Gap

A large part of the available material on this topic consists of framework documentation, blog posts and informal benchmarks. Framework documentation describes features but does not compare frameworks against each other. Blog benchmarks tend to measure trivial "hello world" endpoints, which say very little about a service whose time is dominated by model execution. Many of them also use different hardware, different payloads or different server settings for each framework, so the numbers cannot be compared fairly. Academic work on ML serving, such as systems for low-latency prediction serving and for efficient LLM serving, concentrates on the inference engine and its scheduling, and gives little attention to the choice of the surrounding web framework. As a result, a team that has to choose between FastAPI and Spring Boot for a model API has almost no controlled, reproducible comparison to rely on, and almost nothing that treats classical ML endpoints and streaming LLM endpoints in the same study. This is the gap the present paper addresses.

IV. Problem Statement

Organisations that deploy ML and LLM features must decide whether to serve their models from a Python framework such as FastAPI or from a Java framework such as Spring Boot. At present there is no consistent, reproducible comparison that evaluates both under the same model, the same load and the same deployment conditions, across both quick prediction calls and streaming LLM responses. Without it, the decision is generally made on familiarity or on opinion, which can lead to avoidable cost, poor latency or an architecture that is difficult to maintain.

V. Objectives

1. To study the architecture and concurrency model of Spring Boot and FastAPI from the point of view of ML/LLM inference serving.
2. To design a fair and reproducible benchmark in which both frameworks serve the same model under identical conditions.
3. To compare latency (p50, p95, p99), throughput, CPU usage and memory usage of the two frameworks.
4. To compare start-up time, container image size and operational overhead of the two approaches.
5. To examine how each framework handles streaming responses for LLM-style workloads.
6. To assess developer productivity, ecosystem support and integration effort, and to provide guidelines on when each framework is the better choice.

VI. Proposed Methodology

The study follows an experimental, comparative approach. A single model is served by two independently written services that expose the same REST interface, and both services are tested with the same load generator. The steps are as follows:

7. Model selection and export: a small, well-known model (for example a DistilBERT text classifier or a gradient-boosted tabular model) is chosen and exported to ONNX so that exactly the same model file is used by both services.
8. Service implementation: a FastAPI service and a Spring Boot service are written, each exposing a POST /predict endpoint with the same JSON request and response format, request validation and a health endpoint. Both load the model once at start-up and reuse the session for every request.
9. LLM gateway scenario: each framework also exposes a /generate endpoint that forwards a prompt to the same external inference engine (for example a vLLM or llama.cpp server) and streams the tokens back to the client using Server-Sent Events. This isolates the cost of the web layer from the cost of generation.
10. Controlled environment: both services are packaged as Docker containers with the same CPU and memory limits and are run on the same machine, one at a time, so that neither interferes with the other.
11. Load generation: a tool such as k6 or wrk is used to send requests at increasing concurrency levels (for example 1, 10, 50, 100 and 200 virtual users), after a warm-up period that allows the JVM to finish just-in-time compilation. Each test is repeated several times.
12. Measurement: latency percentiles, requests per second and error rate are taken from the load generator, while CPU and memory are sampled from the container runtime. Start-up time and image size are recorded separately.
13. Analysis: results are averaged over repeated runs, compared in tables and charts, and discussed together with a qualitative assessment of developer experience and ecosystem fit.

VII. Proposed System / Data Sources / Technologies / Tools

The following components are proposed for the experimental set-up:

14. Python 3.11 or later with FastAPI, Pydantic, Uvicorn (and Gunicorn for multi-worker runs) and ONNX Runtime.
15. Java 21 or later with Spring Boot, Spring Web (and optionally Spring WebFlux), and ONNX Runtime for Java.
16. An ONNX model file, shared between both services, together with a fixed set of sample request payloads.

17. An external LLM inference engine (vLLM or llama.cpp server) used for the streaming scenario.
18. Docker for packaging and for enforcing identical resource limits.
19. k6 or wrk for load generation; Prometheus, Grafana or docker stats for resource monitoring.
20. Hardware details (CPU model, cores, RAM, operating system) to be recorded in Table II before the tests begin.

VIII. Comparative Analysis

A. Concurrency and Request Handling

FastAPI achieves concurrency through asynchronous coroutines on a single event loop per worker and through multiple worker processes. Endpoints declared with async def should not run blocking model calls directly, because that would block the event loop; FastAPI therefore runs ordinary def endpoints in a thread pool, and a CPU-bound model call is usually best placed in such an endpoint or in a separate process. Spring Boot with the default servlet stack assigns one thread to each request, which keeps the programming model simple, and with virtual threads this model can handle many more concurrent waiting requests than before. For work that is mostly waiting, such as relaying a streamed LLM response, both approaches work, but they scale in different ways and use memory differently.

B. Model Loading and Inference

In Python the model is normally used in its native form, for example with PyTorch or Hugging Face Transformers, and every new research technique is available on the day it is released. The Java route generally requires exporting the model to ONNX or another portable format, which adds a conversion step and can limit support for unusual operators or custom pre-processing. Tokenisation is a particular point to check: many tokenisers are written for Python, and a Java service must either reimplement them or use a compatible library. On the other hand, once the model is exported, ONNX Runtime executes it with the same native engine from both languages, so the raw inference time is expected to be close and the differences come mainly from the web layer, serialisation and memory behaviour.

C. Streaming for LLM Workloads

LLM responses are produced token by token, and users expect to see text appear as it is generated. FastAPI supports this with streaming responses and generators, and Spring supports it with SseEmitter, with reactive Flux types in WebFlux, and with streaming support in Spring AI. In both cases the web framework mainly acts as a relay, so its overhead is small compared with the generation time; what matters more is how many long-lived connections each one can hold open at once and how much memory each connection consumes.

D. Validation, Security and Observability

FastAPI validates input through Pydantic models and produces interactive API documentation automatically. Spring Boot offers Bean Validation, Spring Security for authentication and authorisation, and Actuator with Micrometer for health checks and metrics, which are ready for production use with very little setup. Many enterprises already have standard Spring-based logging, tracing and security configurations, so reusing them can reduce operational effort. FastAPI can reach a similar level, but usually through additional libraries chosen by the team.

E. Deployment and Resource Footprint

A Python image that includes PyTorch can be large, while a Python image that only needs ONNX Runtime is considerably smaller. A Spring Boot service needs a JVM, and its memory footprint is influenced by heap settings, though modern container-aware JVMs handle limits sensibly. Start-up time matters for autoscaling and serverless use: the JVM generally starts more slowly and needs a warm-up, whereas Python starts quickly but may load a large model before it is ready. GraalVM native images are an option on the Java side for faster start-up, at the cost of a more complicated build.

F. Developer Productivity and Team Fit

Data scientists are normally comfortable with Python, so a FastAPI service can be written, changed and debugged by the same people who built the model. Spring Boot fits teams that already have Java expertise, shared libraries and platform tooling. The cost of having a model passed from one team to another, including the export and the re-implementation of pre-processing, is a real factor and should be counted alongside the benchmark figures.
Table I summarises the qualitative differences discussed above.

TABLE I. Qualitative comparison of FastAPI and Spring Boot for model serving

Aspect	FastAPI (Python)	Spring Boot (Java)
Runtime and server	CPython with Uvicorn / Gunicorn (ASGI)	JVM with embedded Tomcat or Netty
Concurrency model	Async event loop, thread pool, multiple worker processes	Thread per request, virtual threads, or reactive WebFlux
Model format	Native framework models or ONNX	Usually ONNX or another exported format
Request validation	Pydantic, automatic OpenAPI docs	Bean Validation, springdoc for OpenAPI
Streaming	StreamingResponse, async generators	SseEmitter, Flux (WebFlux), Spring AI
Observability	Add-on libraries (Prometheus client, OpenTelemetry)	Actuator and Micrometer built in
Start-up behaviour	Fast process start, depends on model load	Slower start, JIT warm-up needed
Best fit	ML-centric teams, rapid iteration	Enterprise back ends, existing Java platform

IX. Data Analysis / Interpretation / Model Evaluation

The quantitative evaluation will follow the benchmark design described above. Results are to be recorded by the researcher after running the experiments, and are interpreted using the following measures: latency at the 50th, 95th and 99th percentile, sustained throughput in requests per second, error rate, peak CPU utilisation, peak resident memory,

container start-up time to first successful response, and container image size. Tail latency (p95 and p99) is given more weight than the average, because it is what users of an interactive application actually notice. For the streaming scenario, time to first token and the number of simultaneous open streams the service can sustain are recorded as well.
Table II records the test environment and Table III is the result sheet for the main prediction endpoint.

TABLE II. TEST ENVIRONMENT

Parameter	Value
CPU / cores	(to be filled)
RAM	(to be filled)
Operating system	(to be filled)
Container limits	(to be filled)
Python / FastAPI / Uvicorn versions	(to be filled)
Java / Spring Boot versions	(to be filled)
Model and ONNX Runtime version	(to be filled)

TABLE III. RESULT SHEET FOR THE /PREDICT ENDPOINT (TO BE COMPLETED AFTER EXPERIMENTS)

Metric	Concurrency	FastAPI	Spring Boot	Unit	Remarks
Latency p50	50			ms	
Latency p95	50			ms	
Latency p99	50			ms	
Throughput	50			req/s	
Peak CPU	50			%	
Peak memory	50			MB	
Start-up time	–			s	
Image size	–			MB	

Once the table is complete, the discussion should state clearly which framework performed better on each metric, by what margin, and under what load level the difference becomes noticeable. Any result that is surprising, for example a case where the Java service is slower than expected before warm-up or where the Python service saturates early because of the single event loop, should be explained through the concurrency model and not simply reported. Limitations such as the use of a single machine, a single model and a limited number of runs should also be acknowledged.

X. EXPECTED OUTCOMES / FINDINGS

Expected (to be confirmed by the experiments):

21. Because both services call the same native ONNX Runtime engine, raw inference time is expected to be similar, so differences will mostly come from request handling, serialisation and memory behaviour rather than from the model itself.
22. Spring Boot is expected to need a warm-up period but then to behave steadily at high concurrency, while FastAPI is expected to start quickly and to need multiple workers to use all CPU cores.
23. For the LLM gateway scenario, both frameworks are expected to add only a small overhead compared with generation time, so the choice will depend more on connection handling and on the team than on speed.
24. A set of practical guidelines on when to choose each framework, based on workload type, team skills and operational constraints.

Future scope:

25. Adding other frameworks such as Node.js, Go or Rust-based servers for a wider comparison.
26. Testing GPU-backed inference and dynamic batching.
27. Comparing GraalVM native images and Spring WebFlux against the default configuration.
28. Studying cost per million requests on public cloud platforms.

XI. CONCLUSION

Both Spring Boot and FastAPI are capable of serving ML and LLM inference APIs, and neither is the right answer for every situation. FastAPI offers the shortest path from a trained model to a working API and the closest access to the Python ML ecosystem, while Spring Boot offers a mature, production-hardened platform that suits organisations already invested in Java. A fair decision depends on measured behaviour under the real workload, which is why this paper proposes a controlled and repeatable benchmark together with a structured qualitative comparison. The findings from that benchmark, once collected, will give practitioners a concrete basis for choosing the framework instead of relying on preference.

REFERENCES

[1] Ramírez, S. FastAPI documentation. https://fastapi.tiangolo.com
[2] VMware / Broadcom. Spring Boot Reference Documentation. https://docs.spring.io/spring-boot/
[3] ONNX Runtime developers. ONNX Runtime documentation (Python and Java APIs). https://onnxruntime.ai
[4] Amazon Web Services. Deep Java Library (DJL) documentation. https://djl.ai
[5] Paszke, A. et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. Advances in Neural Information Processing Systems 32.
[6] Vaswani, A. et al. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems 30.
[7] Crankshaw, D. et al. (2017). Clipper: A Low-Latency Online Prediction Serving System. 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI).
[8] Kwon, W. et al. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP).
[9] Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.
[10] Python Software Foundation. Global Interpreter Lock, Python documentation. https://docs.python.org
[11] OpenJDK. JEP 444: Virtual Threads. https://openjdk.org/jeps/444

APPENDIX A: MINIMAL ENDPOINT SKETCHES

The two listings below show the shape of the /predict endpoint in each framework. They are starting points and should be adapted to the chosen model (input names, shapes and output types).

A.1 FastAPI

```
import numpy as np
import onnxruntime as ort
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()
session = ort.InferenceSession("model.onnx")
input_name = session.get_inputs()[0].name

class PredictRequest(BaseModel):
    features: list[float]

@app.post("/predict")
def predict(req: PredictRequest):
    x = np.array([req.features],
dtype=np.float32)
    out = session.run(None, {input_name: x})
    return {"prediction": out[0].tolist()}
```

A.2 Spring Boot

```
@RestController
public class PredictController {
  private final OrtEnvironment env =
OrtEnvironment.getEnvironment();
  private final OrtSession session;

  public PredictController() throws
OrtException {
    this.session =
env.createSession("model.onnx", new
OrtSession.SessionOptions());
  }

  public record PredictRequest(float[]
features) {}

  @PostMapping("/predict")
  public Map<String, Object>
predict(@RequestBody PredictRequest req)
throws OrtException {
    String name =
session.getInputNames().iterator().next();
    float[][] input = new float[][] {
req.features() };
    try (OnnxTensor t =
OnnxTensor.createTensor(env, input);
         OrtSession.Result r =
session.run(Map.of(name, t))) {
      return Map.of("prediction",
r.get(0).getValue());
    }
  }
}
```