Serverless Jupyter Notebooks

Comparative Analysis of JupyterLab Deployment Models: Serverless (Knative) vs. Kubernetes Deployment

Michael Pisman

Agenda

- Project Motivation
- Background & Architectures
- Testbed & Configuration
- Performance Metrics & Methodology
- Key Results
- Conclusions & Next Steps

Background: Knative

- CNCF project adding serverless primitives on Kubernetes
- Key features: Serving (scale-to-zero, concurrency), Eventing (CloudEvents), standard CRDs
- Benefits: request-driven autoscaling, scale-to-zero, traffic splitting

Background: Kubernetes Deployment

- Standard Deployment + HPA approach
- Pod replicas based on CPU/memory
- Scale-to-zero manually when needed
- One user per pod; limited density and higher idle cost

Project Motivation

- **Resource Efficiency**: minimize idle consumption and scale-to-zero
- **Latency & UX**: reduce cold-start delays for interactive sessions
- **Comparison Gap**: serverless notebooks vs. traditional one-pod-per-user

Traditional JupyterHub Deployment

- Zero-to-JupyterHub Helm chart on Kubernetes
- KubeSpawner: one pod per user
- Authentication via OAuth/LDAP
- Persistent storage via PVC per user
- Strong isolation but high idle resource cost

Kubernetes Deployment + HPA

- Standard Deployment (apps/v1)
- **HPA**: scales pods on CPU utilization
- **Scale-to-zero**: manual or zero replicas
- **One user** → **one pod** (no request multiplexing)

Jupyter Enterprise Gateway

- Decouples notebook UI from remote kernel execution
- Kernels run on Kubernetes, YARN, Spark clusters
- Single notebook server + remote per-kernel pods
- **Comparison to Serverless Kernel Farm:**
    - Enterprise Gateway: per-kernel pod vs. shared in-pod hosts
    - Enterprise Gateway: multi-tenancy via resource managers; ours: function grouping for density

Serverless JupyterLab (Knative)

- JupyterLab container on Knative Serving
- **Autoscale**: 0 → n pods on HTTP load
- **Buffering**: queue-proxy smooths cold starts
- **Concurrency**: multiple requests per pod

Testbed & Configuration

- **Hardware**: 3-node cluster (DL380 Gen9, 2x E5-2690v4, 128GB RAM)
- **Kubernetes Cluster**: MicroK8s 1.32 HA
- **Knative Serving**: 1.17
- **Images**: same `docker.io/mpisman/jupyterlab:latest`
- **Workload:** cold/warm HTTP GET `/lab?token=test-token`, 10 concurrent requests

Performance Metrics

- **Cold-Start Latency:** request → first "200 OK"
- **Warm-Start Latency:** subsequent requests under active pod(s)
- **Scale-Out Behavior:** pod count over time
- **Resource Utilization:** CPU-sec & memory-MB during test

Methodology

1. **Cold Test:** 0 replicas → single request → record latency
2. **Warm Test:** 1 pod Running → single request → record latency
3. **Load Test:** 10 concurrent requests → observe scaling & latencies
4. **Data Collection:** `curl` timings + `kubectl get pods -w` + Prometheus

Key Findings

Cold-Start Latency

- **Knative:** ~3.19 s
- **HPA:** ?

Warm-Start Latency

- **Knative:** 150 ms
- **HPA:** ?

Scale-Out Under Load

- **Knative:** scaled to 3 pods at 10 RPS
- **HPA:** ?

Conclusions

- **Knative Serverless:** faster cold-starts (3× improvement), zero idle cost, seamless request-driven scaling
- **Kubernetes + HPA:** simpler setup, slower cold-starts, manual scale-to-zero, one pod per user limits density

Next Steps

- **Sticky Sessions:** configure Traefik affinity or shared session store
- **Multi-Tenant Scaling:** prototype serverless kernel farm
- **Extended Metrics:** cell-level latency & cost modeling
- **Demo:** live comparison & Grafana dashboards

Future Plans: Serverless IPyKernel

- Develop custom `KernelManager` for HTTP-based cell execution
- Build in-pod execution broker with process/WASM isolation
- Externalize notebook state to Redis/DB for multi-pod sessions

Tentative Roadmap

- **Phase 1:** prototype `serverless_notebooks` with Knative
- **Phase 2:** integrate into JupyterHub as RemoteKernel
- **Phase 3:** evaluate cell latency, cost, density
- **Phase 4:** secure sandboxing & production hardening

Thank You

Questions?