SPARC Containerization and Deployment

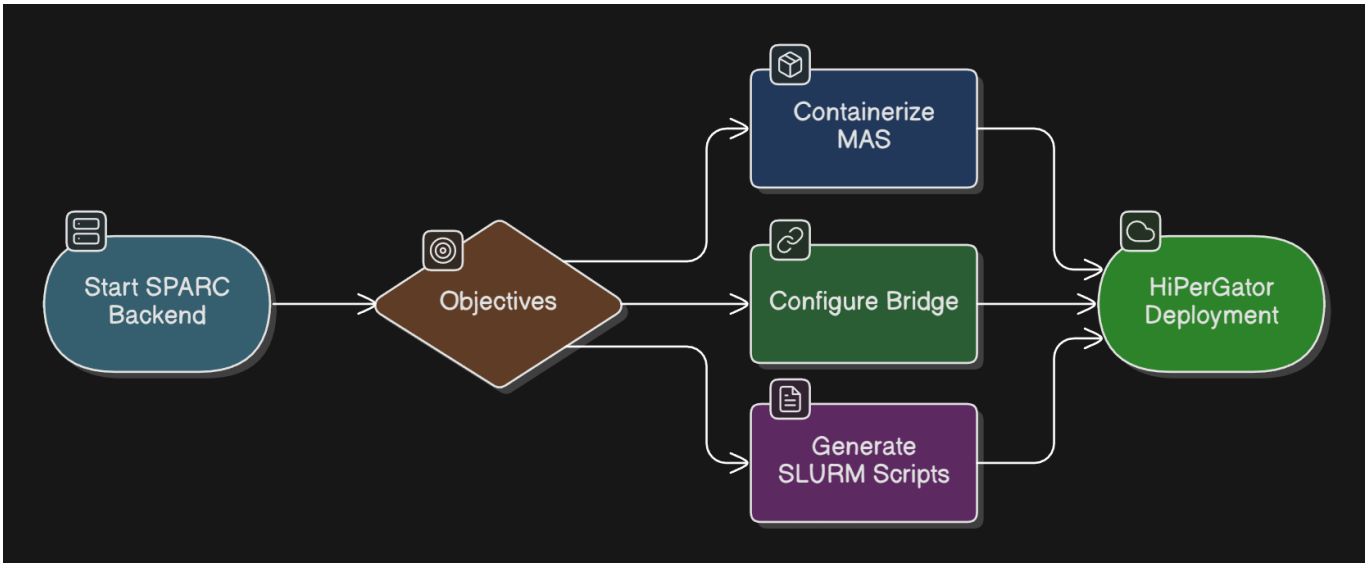
1.0 Introduction

This notebook covers packaging SPARC services as portable containers and preparing both local Podman validation and HiPerGator production deployment artifacts.

1.1 Objectives

1. **Containerize**: Build images for MAS backend, Unity Linux server runtime, and Signaling Server.
2. **Orchestrate**: Validate local Podman pod networking for server-side rendering (Pixel Streaming).
3. **Deploy**: Generate production SLURM artifacts for HiPerGator-compatible backend workflows.

1.2 Introduction Diagram

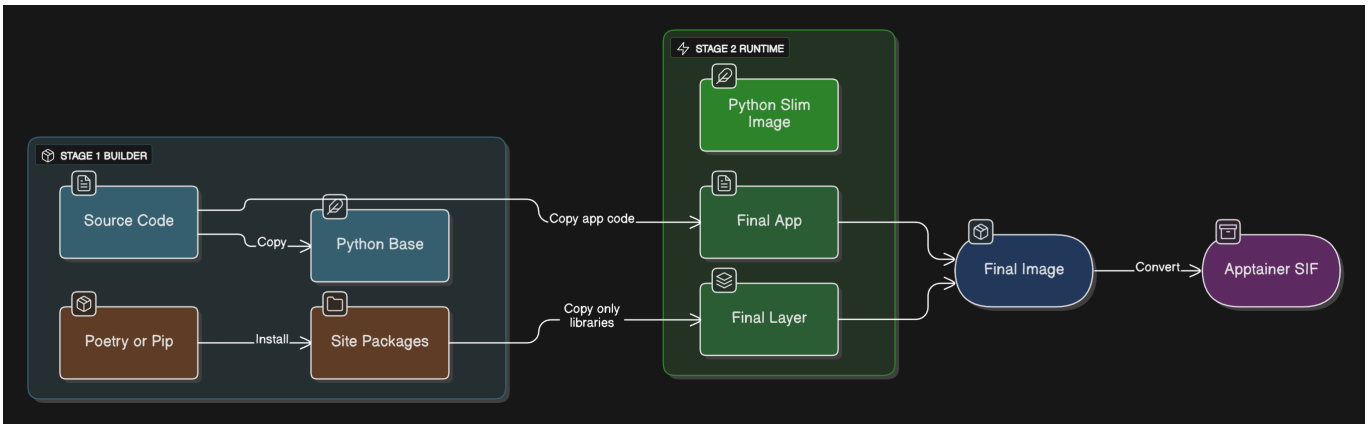


Introduction: This section defines the containerization objectives for the SPARC stack. We now support server-side rendering for thin clients by introducing a Unity Linux server runtime and a signaling container in addition to the backend services.

2.0 Containerization (Docker/Podman -> Apptainer)

We develop with Docker/Podman and deploy with Apptainer on HPC when needed.

2.0 Container Build Strategy Diagram



Container Build Strategy: The flow uses secure, minimal runtime images. Build steps compile dependencies in dedicated stages, then copy only required artifacts into runtime images.

2.1 Dockerfile Definition

This section provides image definitions for three build targets:

1. **MAS Backend** (`Dockerfile.mas`)
2. **Unity Linux Server Build** (`Dockerfile.unity-server`)
3. **WebRTC Signaling Server** (`Dockerfile.signaling`)

2.2 Dockerfile for Multi-Agent System (MAS)

Note on Conda vs Containers: On HiPerGator and PubApps you can deploy with conda environments, but containers are useful for portability and repeatability.

```
# 2.2 Dockerfile for Multi-Agent System (MAS)
def create_mas_dockerfile():
    dockerfile_content = """
# --- Build Stage ---
FROM python:3.11-slim as builder
WORKDIR /app

RUN apt-get update && apt-get install -y --no-install-recommends \\
    build-essential \\
    curl \\
    && rm -rf /var/lib/apt/lists/*

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
COPY . .

# --- Runtime Stage ---
FROM python:3.11-slim
WORKDIR /app

RUN apt-get update && apt-get install -y --no-install-recommends \\
    curl \\
    && rm -rf /var/lib/apt/lists/*
```

```
COPY --from=builder /usr/local/lib/python3.11/site-packages
/usr/local/lib/python3.11/site-packages
COPY --from=builder /app /app

EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
    """
    with open("Dockerfile.mas", "w") as f:
        f.write(dockerfile_content.strip())
    print("Created Dockerfile.mas")

create_mas_dockerfile()
```

2.3 Dockerfile for Unity Linux Server (Render Streaming)

This image packages a Unity Linux player build for server-side rendering.

```
# 2.3 Dockerfile for Unity Linux Server Build
def create_unity_server_dockerfile():
    dockerfile_content = """
FROM nvidia/opengl:1.2-glvnd-runtime-ubuntu20.04

WORKDIR /app

RUN apt-get update && apt-get install -y --no-install-recommends \\
    libnss3 \\
    libxss1 \\
    libasound2 \\
    libglu1-mesa \\
    libxi6 \\
    ca-certificates \\
    && rm -rf /var/lib/apt/lists/*

# Copy Linux server build output from Unity CI artifact
COPY Build/LinuxServer/ /app/

RUN chmod +x /app/SPARC-P.x86_64

# Typical Render Streaming launch flags.
# Use -batchmode/-nographics for headless workflows, or -force-vulkan when package
requires GPU rendering context.
CMD ["/app/SPARC-P.x86_64", "-logFile", "/dev/stdout", "-batchmode", "-force-
vulkan"]
    """
    with open("Dockerfile.unity-server", "w") as f:
        f.write(dockerfile_content.strip())
    print("Created Dockerfile.unity-server")

create_unity_server_dockerfile()
```

2.4 Dockerfile for WebRTC Signaling Server (Node.js)

Use the signaling server from the Unity Render Streaming package.

```
# 2.4 Dockerfile for Signaling Server
def create_signaling_dockerfile():
    dockerfile_content = """
FROM node:20-alpine

WORKDIR /app

# Copy signaling source from Unity Render Streaming package artifact
COPY signaling/ /app/

RUN npm ci --omit=dev

EXPOSE 8080 8888

# 8080: HTTP/WebSocket signaling endpoint
# 8888: Optional metrics/admin endpoint if enabled by your signaling package
CMD ["node", "server.js", "--httpPort", "8080"]
    """
    with open("Dockerfile.signaling", "w") as f:
        f.write(dockerfile_content.strip())
    print("Created Dockerfile.signaling")

create_signaling_dockerfile()
```

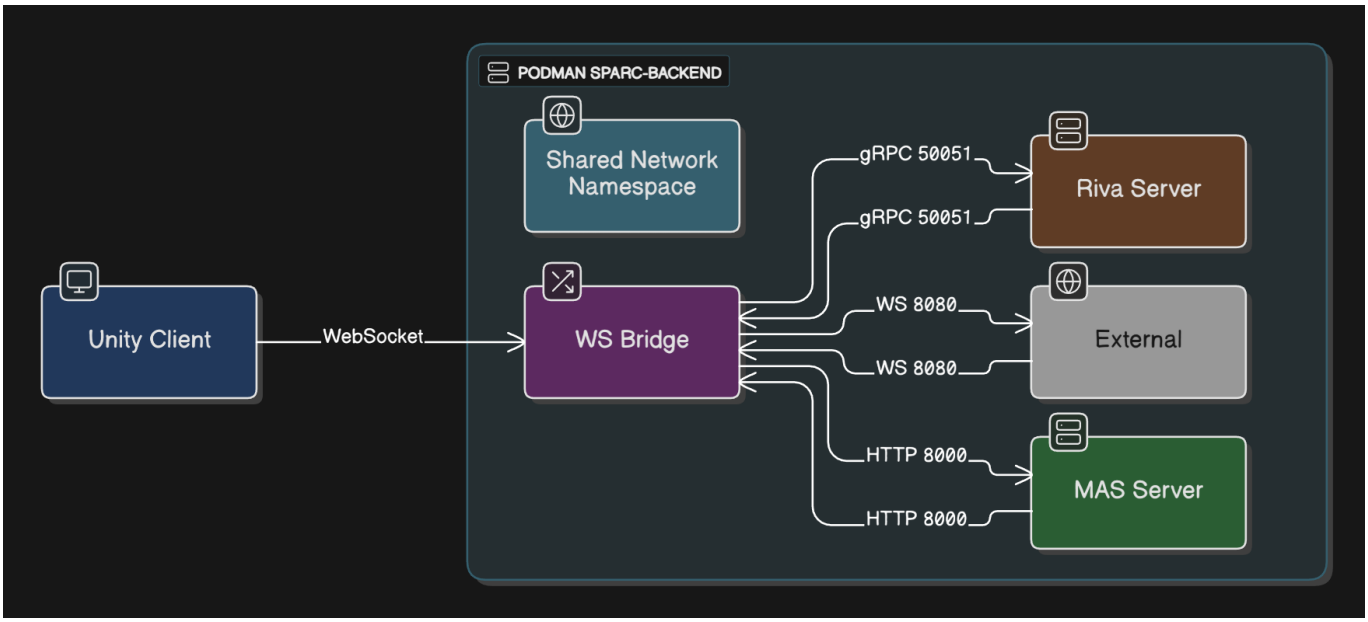
2.5 Build Commands (Reference)

```
podman build -f Dockerfile.mas -t sparc/mas-server:latest .
podman build -f Dockerfile.unity-server -t sparc/unity-server:latest .
podman build -f Dockerfile.signaling -t sparc/signaling-server:latest .
```

3.0 Local Development with Podman

Podman pods allow local validation of the same localhost service mesh used in deployment.

3.0 Local Development Pod Diagram



Local Development Pod (Podman): The local pod now includes MAS backend, Riva, Unity Linux server renderer, and signaling server. The browser receives a live video stream over WebRTC rather than loading a local Unity WebGL build. Audio2Face is removed from this architecture.

3.1 Podman Local Workflow

For local development, run all core services in one pod so they share localhost routing:

- Unity renderer -> localhost services (Riva + backend APIs)
- Signaling server -> browser negotiation path
- Browser -> receives WebRTC stream from Unity runtime

3.2 Podman Workflow (Reference Commands)

```
# 3.2 Podman Workflow (Reference Commands)
podman_commands = """
# 1. Create Pod with signaling + API ports and WebRTC UDP range
podman pod create --name sparc-avatar \\
  -p 8000:8000 \\
  -p 8080:8080 \\
  -p 3478:3478/udp \\
  -p 49152-49200:49152-49200/udp

# 2. Run MAS Server
podman run -d --pod sparc-avatar --name mas-server sparc/mas-server:latest

# 3. Run Riva Server
podman run -d --pod sparc-avatar --name riva-server \\
  --device nvidia.com/gpu=all \\
  nvcr.io/nvidia/riva/riva-speech:2.16.0-server

# 4. Run Unity Linux Render Streaming Server
podman run -d --pod sparc-avatar --name unity-server \\
  --device nvidia.com/gpu=all \\
  -e SIGNALING_URL=ws://localhost:8080 \\
```

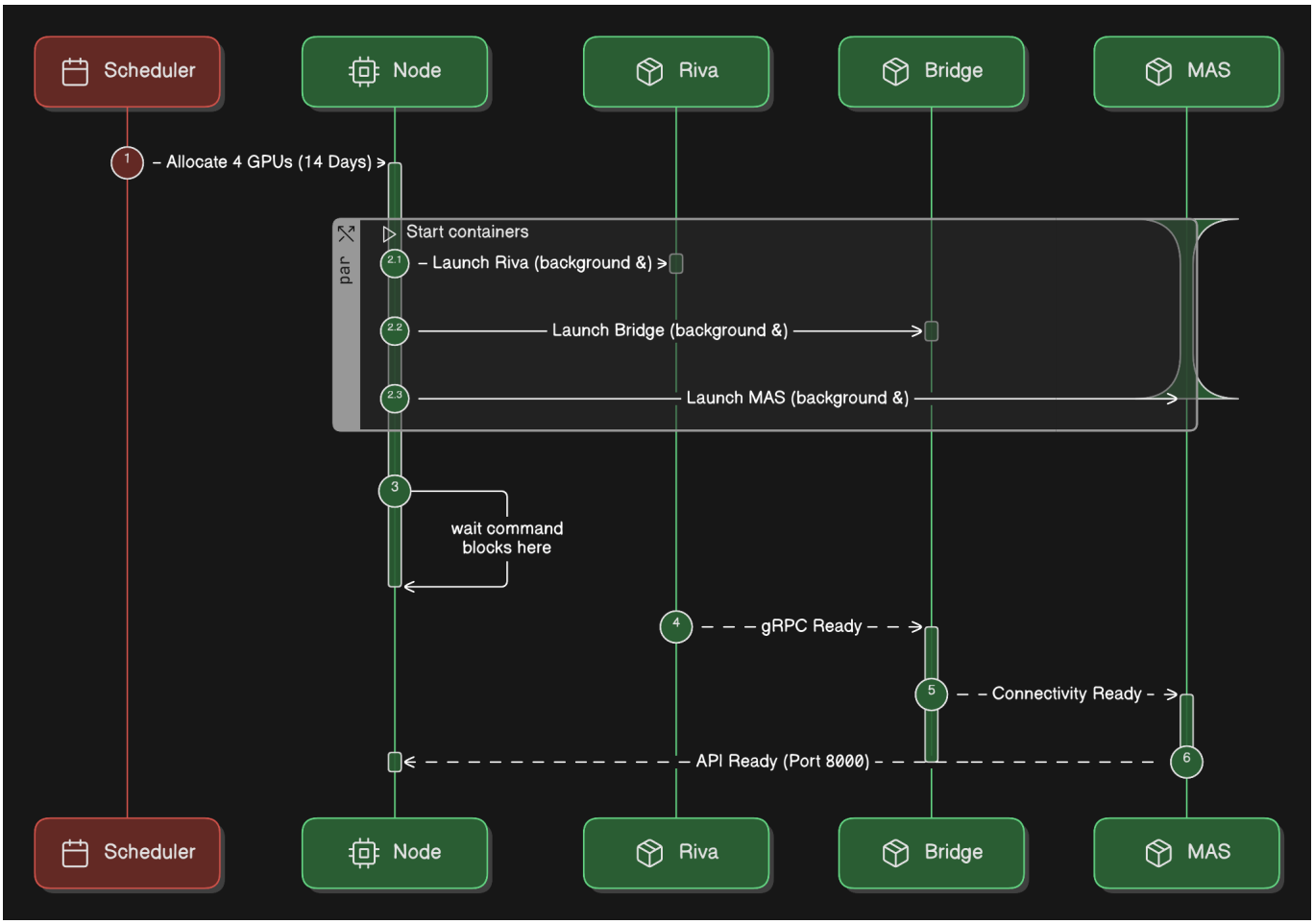
```
    sparc/unity-server:latest

# 5. Run Signaling Server
podman run -d --pod sparc-avatar --name signaling-server \\
    -e PUBLIC_HOST=localhost \\
    sparc/signaling-server:latest
"""
print(podman_commands)
```

4.0 Production Deployment on HiPerGator

Deploy persistent backend services using SLURM and Apptainer where required.

4.0 Production Deployment Diagram



Production Deployment (SLURM): This diagram represents the HiPerGator scheduling path for backend services. PubApps runtime orchestration is handled by Podman + systemd user services.

4.1 Building SIF Images

HiPerGator uses Apptainer, which requires Singularity Image Format (`.sif`) files.

4.2 Build SIF Images

```
# 4.2 Build SIF Images
# !module load apptainer
# !apptainer build mas_server.sif docker-daemon://sparc/mas-server:latest
print("Build SIF images from Docker sources before production deployment on HPG.")
```

4.3 Production Service Launch

This function generates a baseline `sparc_production.slurm` script for backend execution on HiPerGator.

4.4 Production SLURM Script Generator

```
# 4.4 Production SLURM Script Generator
def generate_production_script():
    script_content = """
#!/bin/bash
#SBATCH --job-name=sparc-production-service
#SBATCH --partition=hpg-ai
#SBATCH --nodes=1
#SBATCH --gpus-per-task=4
#SBATCH --cpus-per-task=32
#SBATCH --mem=256gb
#SBATCH --time=14-00:00:00
#SBATCH --output=sparc_service_%j.log

module purge
module load apptainer

MAS_SIF="/blue/jasondeanarnold/SPARCP/containers/mas_server.sif"

echo "Starting MAS..."
apptainer exec --nv ${MAS_SIF} uvicorn main:app --host 0.0.0.0 --port 8000 &

wait
    """
    with open("sparc_production.slurm", "w") as f:
        f.write(script_content.strip())
    print("Generated sparc_production.slurm")

generate_production_script()
```

Summary

This notebook now covers both classic backend containerization and the new server-side rendering container artifacts:

1. **MAS Backend Container** for API/model orchestration.
2. **Unity Linux Server Container** for GPU-rendered server-side scene streaming.
3. **Node Signaling Container** for WebRTC negotiation.

4. **Podman Pod Topology** for localhost service routing and browser video streaming.
5. **HiPerGator Production Script** for backend workflows where Apptainer + SLURM are required.

The architecture removes Audio2Face from the deployment path and aligns local container artifacts with the PubApps Pixel Streaming runtime.