SPARC-P PubApp Deployment Guide

Overview

This notebook provides step-by-step instructions for deploying the SPARC-P backend to **UF RC PubApps** for public access. PubApps is a separate infrastructure from HiPerGator designed for hosting web applications that serve research results.

Key Differences: HiPerGator vs PubApps

Aspect	HiPerGator	PubApps
Purpose	Model training & batch processing	Public web application hosting
Access	Internal (UF network + VPN)	Public internet
Containers	Apptainer only	Podman only
Storage	`/blue` (shared with HiPerGator)	`/pubapps` (1TB included)
Scheduling	SLURM batch jobs	Systemd services (persistent)
GPUs	4 GPUs available for parallelization	1x L4 (24GB) for inference
CPU / RAM	16 CPU cores available	2 CPU cores, 16GB RAM
Conda	Yes, via modules	Yes, can be installed

1.0 Prerequisites

1.1 Required Allocations

Before deploying to PubApps, ensure you have:

1. **PA-Instance allocation** ($300/year) - Request via HiPerGator Service Purchase Form
2. **PA-GPU allocation** (if using GPU inference) - L4 GPUs available
3. **Completed Risk Assessment** - See Risk Assessment Documentation

1.2 PubApp Instance Setup

After purchasing a PA-Instance, open a support ticket to provision your instance:

- Instance will be accessible via SSH from HiPerGator
- You'll receive a project user account (usually matches your HiPerGator group name)
- Default resources for this project: 1x L4 (24GB), 2 vCPUs, 16GB RAM, 1TB `/pubapps` storage

1.3 Architecture Overview

```
┌──────────────────────────────────────────────────────────────┐
│                       Public Internet                        │
│                          │                          │
│                   UF Shibboleth                     │
│                  (Authentication)                   │
└──────────────────────────┬──────────────────────────┘
                           │
                           │
┌──────────────────────────▼──────────────────────────┐
│                  PubApps Instance                   │
│  ┌───────────────────────────────────────────────┐  │
│  │             NGINX Reverse Proxy               │  │
│  │         (SSL termination, routing)            │  │
│  └─────────┬──────────────────────────┬──────────┘  │
│            │                          │             │
│  ┌─────────▼─────────┐    ┌───────────▼───────────┐ │
│  │  Unity WebGL      │    │  FastAPI Backend      │ │
│  │  (Static Files)   │    │  (Conda Environment)  │ │
│  └───────────────────┘    │                       │ │
│                           │  ┌─────────────────┐  │ │
│                           │  │  LangGraph      │  │ │
│                           │  │  Orchestration  │  │ │
│                           │  └─────────────────┘  │ │
│                           │  ┌─────────────────┐  │ │
│                           │  │ Trained Models  │  │ │
│                           │  │ (from /blue)    │  │ │
│                           │  └─────────────────┘  │ │
│                           └──────────┬────────────┘ │
│                                      │              │
│                           ┌──────────▼───────────┐  │
│                           │  Riva Container      │  │
│                           │  (ASR/TTS, L4 GPU)   │  │
│                           └──────────────────────┘  │
│                                      │              │
│  ┌────────────────────────────────┘              │
│  │          Firebase Firestore (External)        │  │
│  │          (Session state, metrics)             │  │
│  └───────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────┘
```

2.0 Transfer Trained Models from HiPerGator

2.1 Sync Models to PubApps Storage

PubApps storage (`/pubapps`) is NOT directly accessible from HiPerGator. You must transfer files.

```
# On HiPerGator, after training completes
# Models are in: /blue/jasondeanarnold/SPARCP/trained_models

# Method 1: Use rsync from HiPerGator to PubApps
# (Requires SSH access to PubApps instance - you must SSH through HPG first)
rsync -avz --progress \
  /blue/jasondeanarnold/SPARCP/trained_models/ \
    SPARCP@pubapps-vm.rc.ufl.edu:/pubapps/SPARCP/models/

# Method 2: Use Globus (recommended for large models)
# Set up Globus endpoints for both HiPerGator and PubApps
# Transfer via Globus web interface

# Method 3: Stage to intermediate location
# Use /blue/jasondeanarnold/SPARCP as staging
# Then scp/rsync from there
```

2.2 Verify Model Transfer

```
# SSH to PubApps instance (from HiPerGator)
ssh SPARCP@pubapps-vm.rc.ufl.edu

# Check models arrived
ls -lh /pubapps/SPARCP/models/
# Should see: CaregiverAgent/, C-LEAR_CoachAgent/, SupervisorAgent/
```

3.0 Setup Conda Environment on PubApps

3.1 Install Conda on PubApps

PubApps VMs don't have the `module` system like HiPerGator. Install miniconda directly:

```
# SSH to PubApps instance
ssh SPARCP@pubapps-vm.rc.ufl.edu

# Download miniconda
cd ~
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh

# Install (accept license, use default location: ~/miniconda3)
bash Miniconda3-latest-Linux-x86_64.sh -b -p ~/miniconda3

# Initialize conda for bash
~/miniconda3/bin/conda init bash
source ~/.bashrc

# Verify installation
conda --version
```

3.2 Create Backend Environment

```
# On PubApps VM
cd /pubapps/SPARCP

# Transfer environment file from HiPerGator notebooks
# Option 1: Copy from HiPerGator
scp jayrosen@hpg.rc.ufl.edu:/path/to/Sparc\ Hipergator\
Notebooks/environment_backend.yml .

# Option 2: Create manually using the environment_backend.yml from the notebooks
repo

# Create environment in /pubapps to avoid home directory space issues
conda env create -f environment_backend.yml -p
/pubapps/SPARCP/conda_envs/sparc_backend

# Activate environment
conda activate /pubapps/SPARCP/conda_envs/sparc_backend

# Verify installation
python -c "import torch; print(f'PyTorch: {torch.__version__}'); print(f'CUDA:
{torch.cuda.is_available()}')"
python -c "import fastapi, langgraph, transformers; print('✓ All packages
available')"
```

4.0 Deploy Riva Speech Services

4.1 Pull Riva Container with Podman

```
# On PubApps VM (Podman is pre-installed, NOT Docker)
# Note: Use podman, not docker commands

# Pull Riva server image
podman pull nvcr.io/nvidia/riva/riva-speech:2.16.0-server

# Create persistent storage for Riva models
mkdir -p /pubapps/SPARCP/riva_models

# Initialize Riva (downloads ASR/TTS models, ~10GB)
# This requires GPU access - run with --hooks-dir option for rootless podman
podman run --rm -it \
  --gpus all \
    -v /pubapps/SPARCP/riva_models:/data \
  nvcr.io/nvidia/riva/riva-speech:2.16.0-server \
  bash -c "cd /data && /opt/riva/bin/riva_init.sh"
```

4.2 Configure Riva Server

Create a Podman Quadlet service file for systemd integration:

```
# Create systemd user service directory
mkdir -p ~/.config/containers/systemd

# Create Riva service file
cat > ~/.config/containers/systemd/riva-server.container << 'EOF'
[Unit]
Description=SPARC-P Riva Speech Server
After=network-online.target

[Container]
Image=nvcr.io/nvidia/riva/riva-speech:2.16.0-server
ContainerName=riva-server
# GPU access
AddDevice=/dev/nvidia0
AddDevice=/dev/nvidiactl
AddDevice=/dev/nvidia-uvm
# Volume mounts
Volume=/pubapps/SPARCP/riva_models:/data:Z
# Network
PublishPort=50051:50051
# Environment
Environment=NVIDIA_VISIBLE_DEVICES=all
# Command
Exec=/opt/riva/bin/riva_server --riva_model_repo=/data/models

[Service]
Restart=always
TimeoutStartSec=300

[Install]
WantedBy=default.target
EOF

# Reload systemd
systemctl --user daemon-reload

# Enable and start Riva service
systemctl --user enable --now riva-server

# Check status
systemctl --user status riva-server
```

5.0 Deploy FastAPI Backend Service

5.1 Create Backend Application

```
# On PubApps VM
cd /pubapps/SPARCP
```

```
mkdir -p backend
cd backend
```

Create the main FastAPI application (`main.py`):

```
# /pubapps/SPARCP/backend/main.py
"""
SPARC-P FastAPI Backend for PubApps
Serves the trained multi-agent system for public access
"""
import os
import sys
import base64
from typing import Optional
from fastapi import FastAPI, HTTPException, Request
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import PeftModel
import riva.client
from langgraph.graph import StateGraph
import firebase_admin
from firebase_admin import credentials, firestore

# Configuration
MODEL_BASE_PATH = "/pubapps/SPARCP/models"
RIVA_SERVER = "localhost:50051"
FIREBASE_CREDS = "/pubapps/SPARCP/config/firebase-credentials.json"

# Initialize Firebase
cred = credentials.Certificate(FIREBASE_CREDS)
firebase_admin.initialize_app(cred)
db = firestore.client()

# Initialize FastAPI
app = FastAPI(
    title="SPARC-P Multi-Agent Backend",
    description="HPV Vaccine Communication Training System",
    version="1.0.0"
)

# Enable CORS for Unity WebGL
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Configure appropriately for production
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Load models at startup
```

```
@app.on_event("startup")
async def load_models():
    """Load trained agent models into memory"""
    global caregiver_model, coach_model, supervisor_model, tokenizer

    print("Loading base model and adapters...")
    base_model_name = "gpt-oss-120b"  # Adjust to your base model

    # Load tokenizer
    tokenizer = AutoTokenizer.from_pretrained(base_model_name)

    # Load base model (quantized for efficiency)
    base_model = AutoModelForCausalLM.from_pretrained(
        base_model_name,
        load_in_4bit=True,
        device_map="auto"
    )

    # Load agent adapters
    caregiver_model = PeftModel.from_pretrained(
        base_model,
        os.path.join(MODEL_BASE_PATH, "CaregiverAgent")
    )
    coach_model = PeftModel.from_pretrained(
        base_model,
        os.path.join(MODEL_BASE_PATH, "C-LEAR_CoachAgent")
    )
    supervisor_model = PeftModel.from_pretrained(
        base_model,
        os.path.join(MODEL_BASE_PATH, "SupervisorAgent")
    )

    print("✓ Models loaded successfully")

# Pydantic models
class ChatRequest(BaseModel):
    session_id: str
    user_message: str
    audio_data: Optional[str] = None  # Base64 encoded audio

class ChatResponse(BaseModel):
    response_text: str
    audio_url: Optional[str] = None
    emotion: str
    animation_cues: dict
    coach_feedback: Optional[dict] = None

# Health check endpoint
@app.get("/health")
async def health_check():
    """Health check for monitoring"""
    try:
        auth = riva.client.Auth(uri=RIVA_SERVER)
        riva.client.ASRService(auth)
        riva_ok = True
    except Exception:
        riva_ok = False

    return {
        "status": "healthy",
        "models_loaded": True,
        "riva_connected": riva_ok
    }

# Main chat endpoint
@app.post("/v1/chat", response_model=ChatResponse)
async def process_chat(request: ChatRequest):
    """
    Process user input through multi-agent system
    """
    try:
        # 1. Retrieve session state from Firestore
        session_ref = db.collection('sessions').document(request.session_id)
        session_state = session_ref.get().to_dict() or {}

        # 2. Process through supervisor (safety check)
        disallowed_topics = ["politics", "election", "gambling", "crypto",
"finance advice"]
        lowered_text = request.user_message.lower()
        if any(topic in lowered_text for topic in disallowed_topics):
            return ChatResponse(
                response_text="I can only discuss topics related to HPV
vaccination and clinical communication training.",
                emotion="neutral",
                animation_cues={"gesture": "idle"},
                coach_feedback={"safe": False, "reason": "off_topic"}
            )

        # 3. Route to appropriate agent (LangGraph-compatible state routing)
        conversation_mode = session_state.get("mode", "caregiver")
        if conversation_mode == "coach":
            active_model = coach_model
        elif conversation_mode == "supervisor":
            active_model = supervisor_model
        else:
            active_model = caregiver_model

        # 4. Generate response (adapter-based inference)
        prompt = f"[SESSION: {request.session_id}] User:
{request.user_message}\nAssistant:"
        model_inputs = tokenizer(prompt, return_tensors="pt", truncation=True,
max_length=1024)
        model_inputs = {k: v.to(active_model.device) for k, v in
model_inputs.items()}

        with torch.inference_mode():
            output_tokens = active_model.generate(
                **model_inputs,
                max_new_tokens=180,
                do_sample=True,
                temperature=0.7,
                top_p=0.9,
                pad_token_id=tokenizer.eos_token_id
            )

        decoded = tokenizer.decode(output_tokens[0], skip_special_tokens=True)
        response_text = decoded.split("Assistant:")[-1].strip() or "I’m here to help with HPV vaccine communication practice."

        # Optional coach feedback generated with coach adapter
        feedback_prompt = f"Provide concise coaching feedback for this response: {response_text}"
        feedback_inputs = tokenizer(feedback_prompt, return_tensors="pt", truncation=True, max_length=512)
        feedback_inputs = {k: v.to(coach_model.device) for k, v in feedback_inputs.items()}
        with torch.inference_mode():
            feedback_tokens = coach_model.generate(
                **feedback_inputs,
                max_new_tokens=80,
                do_sample=False,
                pad_token_id=tokenizer.eos_token_id
            )
        coach_feedback_text = tokenizer.decode(feedback_tokens[0], skip_special_tokens=True)

        # 5. Convert to speech with Riva TTS
        audio_url = None
        try:
            auth = riva.client.Auth(uri=RIVA_SERVER)
            tts_service = riva.client.SpeechSynthesisService(auth)
            tts_response = tts_service.synthesize(response_text, voice_name="English-US.Female-1")
            audio_b64 = base64.b64encode(tts_response.audio).decode("utf-8")
            audio_url = f"data:audio/wav;base64,{audio_b64}"
        except Exception as riva_error:
            print(f"Riva TTS unavailable: {riva_error}")

        # 6. Update session state in Firestore
        session_state["last_user_message"] = request.user_message
        session_state["last_response"] = response_text
        session_state["mode"] = conversation_mode
        session_ref.set(session_state, merge=True)

        return ChatResponse(
            response_text=response_text,
            audio_url=audio_url,
            emotion="supportive",
            animation_cues={"gesture": "speaking", "intensity": "low"},
            coach_feedback={"summary": coach_feedback_text[:500], "safe": True}
        )
```

```
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

# For development only
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

5.2 Create Systemd Service

Create a systemd user service to run the FastAPI backend:

```
# Create service file
cat > ~/.config/systemd/user/sparc-backend.service << 'EOF'
[Unit]
Description=SPARC-P FastAPI Backend
After=network.target riva-server.service
Requires=riva-server.service

[Service]
Type=simple
Environment="PATH=/pubapps/SPARCP/conda_envs/sparc_backend/bin:/usr/bin"
Environment="PYTHONUNBUFFERED=1"
WorkingDirectory=/pubapps/SPARCP/backend
ExecStart=/pubapps/SPARCP/conda_envs/sparc_backend/bin/uvicorn main:app --host
0.0.0.0 --port 8000 --workers 1
Restart=always
RestartSec=10

[Install]
WantedBy=default.target
EOF

# Reload systemd
systemctl --user daemon-reload

# Enable and start service
systemctl --user enable --now sparc-backend

# Check status
systemctl --user status sparc-backend

# View logs
journalctl --user -u sparc-backend -f
```

6.0 Configure NGINX Reverse Proxy

6.1 Request NGINX Configuration

Open a support ticket to request NGINX reverse proxy configuration:

```
Subject: NGINX Configuration for SPARC-P PubApps Instance

Body:
Please configure NGINX reverse proxy for the SPARC-P application:

1. SSL Certificate: Request *.rc.ufl.edu certificate or custom domain
2. Proxy Rules:
    - / → Unity WebGL static files (/pubapps/SPARCP/unity_webgl/)
   - /api/ → FastAPI backend (http://localhost:8000)
3. WebSocket Support: Enable for /api/ws
4. Authentication: UF Shibboleth SSO for access control
5. CORS: Allow for Unity WebGL

Public URL: https://sparc-p.rc.ufl.edu (or assigned domain)
```

6.2 NGINX Configuration (Reference)

The RC team will configure NGINX, but for reference:

```nginx
server {
    listen 443 ssl http2;
    server_name sparc-p.rc.ufl.edu;

    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;

    # Static Unity WebGL files
    location / {
        root /pubapps/SPARCP/unity_webgl;
        index index.html;
        try_files $uri $uri/ /index.html;
    }

    # API proxy to FastAPI backend
    location /api/ {
        proxy_pass http://localhost:8000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    # WebSocket support
    location /api/ws {
        proxy_pass http://localhost:8000/ws;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        }
    }
```

7.0 Deployment Checklist

7.1 Pre-Deployment

- [ ] PubApps instance provisioned
- [ ] Models transferred from HiPerGator to `/pubapps/SPARCP/models/`
- [ ] Conda environment created and tested
- [ ] Firebase credentials configured
- [ ] UF RC risk assessment completed

7.2 Service Deployment

- [ ] Riva container running (`systemctl --user status riva-server`)
- [ ] FastAPI backend running (`systemctl --user status sparc-backend`)
- [ ] Services set to auto-start on boot
- [ ] Logs configured and accessible

7.3 Integration

- [ ] NGINX reverse proxy configured
- [ ] SSL certificate installed
- [ ] UF Shibboleth SSO configured
- [ ] Unity WebGL build deployed to `/pubapps/SPARCP/unity_webgl/`
- [ ] CORS configured correctly

7.4 Testing

- [ ] Health check endpoint accessible: `https://sparc-p.rc.ufl.edu/api/health`
- [ ] Chat endpoint functional: `POST https://sparc-p.rc.ufl.edu/api/v1/chat`
- [ ] Speech-to-text working (Riva ASR)
- [ ] Text-to-speech working (Riva TTS)
- [ ] Firebase connectivity confirmed
- [ ] End-to-end Unity → Backend → Unity flow tested

8.0 Monitoring and Maintenance

8.1 Service Management

```
# Check service status
systemctl --user status riva-server
systemctl --user status sparc-backend

# View logs
```

```
journalctl --user -u riva-server -n 100
journalctl --user -u sparc-backend -n 100 -f

# Restart services
systemctl --user restart riva-server
systemctl --user restart sparc-backend

# Stop services
systemctl --user stop sparc-backend riva-server
```

8.2 Resource Monitoring

```
# Check disk usage
df -h /pubapps/SPARCP

# Check GPU usage
nvidia-smi

# Check memory usage
free -h

# Check service resource usage
systemctl --user status sparc-backend
```

8.3 Update Workflow

To update models or code:

1. Train new model on HiPerGator
2. Transfer to PubApps via rsync/Globus
3. Restart backend service: `systemctl --user restart sparc-backend`
4. Monitor logs for successful reload

9.0 Security and Compliance

9.1 Data Classification

Per the PubApp request form, SPARC-P processes:

- **Open Data**: Training materials, public health information
- **Sensitive Data**: Session metadata (transient, in-memory processing)
- **Not Stored**: PHI, FERPA records, credentials (handled by Shibboleth)

9.2 Security Controls

1. **Authentication**: UF Shibboleth SSO (delegated to UF IdP)
2. **Transport Security**: TLS 1.2+ for all connections
3. **Data Retention**: Transient PHI model (in-memory only, not persisted)

4. **Audit Logging**: Non-sensitive metrics stored in Firebase Firestore
5. **Access Control**: Limited to authorized project members via SSH keys

9.3 Compliance

- Follow UFIT RC PubApps Policy
- Quarterly user access reviews (document in ticket)
- Vulnerability scanning compliance (RC team performs scans)
- No Critical/High vulnerabilities in production

10.0 Troubleshooting

10.1 Common Issues

Issue: Conda environment not found

```
# Solution: Verify path and activate
conda info --envs
conda activate /pubapps/SPARCP/conda_envs/sparc_backend
```

Issue: Riva container won't start

```
# Solution: Check GPU access and permissions
podman run --rm --gpus all nvidia/cuda:12.8.0-base-ubuntu22.04 nvidia-smi
# Check service logs
journalctl --user -u riva-server -n 50
```

Issue: FastAPI service crashes

```
# Solution: Check logs for errors
journalctl --user -u sparc-backend -n 100
# Verify conda environment packages
conda list | grep -E "fastapi|uvicorn|torch"
```

Issue: "Connection refused" from Unity

```
# Solution: Verify NGINX proxy and backend are running
curl http://localhost:8000/health  # Should return {"status": "healthy"}
# Check NGINX logs (contact RC if needed)
```

10.2 Support Resources

- **UF RC Support Ticket**: https://support.rc.ufl.edu/

- **PubApps Documentation**: https://docs.rc.ufl.edu/services/web_hosting/
- **RC Consulting**: Schedule via support ticket
- **Project Team**: Contact Jason Arnold (jda@coe.ufl.edu)

11.0 Summary

This notebook covered the complete PubApps deployment workflow:

1. ✅ Transferred trained models from HiPerGator to PubApps
2. ✅ Set up conda environment on PubApps VM
3. ✅ Deployed Riva speech services with Podman
4. ✅ Created FastAPI backend with systemd service
5. ✅ Configured NGINX reverse proxy
6. ✅ Implemented security controls (Shibboleth SSO, TLS)
7. ✅ Established monitoring and maintenance procedures

Key Takeaways:

- PubApps is separate from HiPerGator (different infrastructure, access, tools)
- Use **conda** for Python environment management (UF RC requirement)
- Use **Podman** for containers (NOT Docker)
- Use **systemd** for persistent service management
- Models trained on HiPerGator → Deployed on PubApps → Served publicly

Next Steps:

1. Complete PubApps instance request and risk assessment
2. Transfer trained models from HiPerGator
3. Follow deployment steps in this notebook
4. Test end-to-end integration with Unity WebGL
5. Monitor services and maintain documentation