Complete WSL2 + Docker Desktop + VS Code Setup Guide

Machine: DESKTOP-6NMJEGK
CPU: Intel i7-6500U (2 cores, 4 threads)
RAM: 12GB
Storage: 1.82TB HDD
OS: Windows 10 Build 19045.6466
WSL: Version 2.6.3.0

Table of Contents

System Overview

Architecture

```
└── WSL2 (Ubuntu)
    ├── Resource Limits: 6GB RAM, 3 CPUs
    ├── Systemd: Enabled
    │
    └── /home/username/projects/ (ALL CODE HERE)
        ├── web-apps/
        ├── microservices/
        ├── ml-models/
        ├── databases/
        ├── mcp-services/
        ├── _templates/
        ├── _shared/
        └── _docs/
```

Why This Architecture?

1. **Performance:** WSL2 filesystem is 2-5x faster than Windows filesystem for I/O operations
2. **Portability:** Containers work identically across your laptop and desktop
3. **Resource Control:** Prevents Docker from consuming all system resources
4. **Isolation:** Each project in separate container, no dependency conflicts
5. **Git-Friendly:** All code in WSL2 avoids line-ending and permission issues

---

Prerequisites

Required Software

1. **Windows 10/11** (Build 19041 or higher)
2. **WSL2** (Version 0.67.6 or higher)
3. **Docker Desktop for Windows**
4. **VS Code** with extensions:
    - Remote - WSL
    - Docker
    - Python
    - Dev Containers
5. **Git for Windows** (optional, we'll use Git in WSL)

Check Your Current Setup

```powershell
# Run in PowerShell (Windows side)
wsl --version
wsl --status
docker --version
```

Step-by-Step Installation

Phase 1: WSL2 Configuration (Windows Side)

1.1 Create `.wslconfig`

Location: `C:\Users\<YourUsername>\.wslconfig`

```powershell
# Navigate to your home directory
cd ~

# Create .wslconfig file
notepad .wslconfig
```

Paste this content:

```ini
[wsl2]
memory=6GB
processors=3
swap=2GB
localhostForwarding=true
networkingMode=mirrored
dnsTunneling=true
autoProxy=true
sparseVhd=true
nestedVirtualization=true
pageReporting=true
autoMemoryReclaim=gradual
```

Save and close.

1.2 Apply `.wslconfig` Changes

```powershell
# Shutdown WSL (applies new settings)
wsl --shutdown

# Wait 10 seconds, then start WSL
wsl
```

1.3 Verify WSL2 Settings

```powershell
wsl --status
```

Expected output should show WSL2 as default version.

Phase 2: WSL2 Ubuntu Configuration (Linux Side)

Open WSL2 Ubuntu terminal and continue:

2.1 Create `wsl.conf`

```bash
# Edit wsl.conf (requires sudo)
sudo nano /etc/wsl.conf
```

Paste this content:

```ini
```

```
[boot]
systemd=true

[automount]
enabled=true
root=/mnt/
mountFsTab=true
options="metadata,umask=022,fmask=11,dmask=022,case=off"

[network]
generateHosts=true
generateResolvConf=true

[interop]
enabled=true
appendWindowsPath=true
```

Save: Ctrl+X, then Y, then Enter

2.2 Restart WSL2

```bash
# Exit WSL
exit
```

```powershell
# In PowerShell
wsl --shutdown

# Wait 10 seconds
wsl
```

2.3 Verify Systemd

```bash
# Check if systemd is running
systemctl --version

# Should show systemd version info
```

Phase 3: Docker Installation

3.1 Install Docker in WSL2

```bash
# Update package index
sudo apt-get update

# Install prerequisites
sudo apt-get install -y \
    ca-certificates \
    curl \
    gnupg \
    lsb-release

# Add Docker GPG key
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
    sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

# Set up repository
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install Docker
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin

# Add user to docker group
sudo usermod -aG docker $USER

# Apply group changes
newgrp docker
```

3.2 Verify Docker Installation

```bash
```

```
docker --version
docker compose version

# Test Docker (should run without sudo)
docker run hello-world
```

Phase 4: Docker Desktop Configuration (Windows)

4.1 Install Docker Desktop

Download from: https://www.docker.com/products/docker-desktop

4.2 Configure Docker Desktop

1. Open **Docker Desktop**
2. Go to **Settings → General**:
   - ☑ Use WSL 2 based engine
   - ☐ Start Docker Desktop when you log in (disable for manual control)
   - ☐ Open Docker Dashboard at startup
3. Go to **Settings → Resources → WSL Integration**:
   - ☑ Enable integration with my default WSL distro
   - ☑ Ubuntu (or your distro name)
4. Go to **Settings → Resources → Advanced**:
   - **Memory:** 4 GB
   - **CPUs:** 2
   - **Disk image size:** 100 GB
5. Click **Apply & Restart**

4.3 Configure Docker Daemon (Optional)

Location: `%USERPROFILE%\.docker\daemon.json`

Create or edit this file:

```
json
```

```
{
  "builder": {
    "gc": {
      "enabled": true,
      "defaultKeepStorage": "10GB"
    }
  },
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  },
  "default-ulimits": {
    "nofile": {
      "Hard": 64000,
      "Soft": 64000
    }
  }
}
```

Restart Docker Desktop after editing.

Phase 5: Project Directory Setup

5.1 Create Directory Structure

```bash
# Create project directories
mkdir -p ~/projects/{web-apps,microservices,ml-models,databases,mcp-services,_templates,_shared,_docs}

# Create shared resources
mkdir -p ~/projects/_shared/{docker-networks,configs,scripts,volumes}

# Navigate to projects
cd ~/projects
```

5.2 Create Shared Docker Networks

```bash
```

```
# Create networks configuration
nano ~/projects/_shared/docker-networks/docker-compose.yml
```

Paste:

```yaml
version: '3.8'

networks:
  app-network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.20.0.0/16

  db-network:
    driver: bridge
    internal: true

  mcp-network:
    driver: bridge
```

Save and exit.

5.3 Initialize Shared Networks

```bash
cd ~/projects/_shared/docker-networks
docker compose up -d
```

Phase 6: VS Code Setup

6.1 Install VS Code Extensions

1. Open VS Code
2. Install these extensions:
   - **Remote - WSL** (ms-vscode-remote.remote-wsl)
   - **Docker** (ms-azuretools.vscode-docker)

- **Python** (ms-python.python)
- **Dev Containers** (ms-vscode-remote.remote-containers)
- **GitLens** (eamodio.gitlens)

6.2 Connect VS Code to WSL2

```bash
# In WSL terminal
code ~/projects
```

This opens VS Code in WSL mode.

6.3 Configure VS Code Settings

Create: `~/projects/.vscode/settings.json`

```json
{
  "remote.WSL.fileWatcher.polling": true,
  "remote.WSL.useShellEnvironment": true,
  "files.watcherExclude": {
    "**/.git/objects/**": true,
    "**/node_modules/**": true,
    "**/__pycache__/**": true
  }
}
```

Configuration Files

Project Template Structure

Each project should have this structure:

```
my-project/
├── .devcontainer/
│   └── devcontainer.json      # VS Code dev container config
├── .vscode/
│   ├── settings.json          # Editor settings
│   └── launch.json            # Debug configurations
├── docker/
│   └── init-db/               # Database initialization scripts
├── src/
│   ├── __init__.py
│   └── main.py
├── tests/
│   └── test_main.py
├── .gitignore                 # Git exclusions
├── .dockerignore              # Docker build exclusions
├── .env.example               # Environment template
├── docker-compose.yml         # Service orchestration
├── Dockerfile                 # Container definition
├── requirements.txt           # Python dependencies
└── README.md
```

Essential Files

All template files are provided in the artifacts above:

1. `.gitignore` - Universal Python/Docker exclusions
2. `.dockerignore` - Optimized build context
3. `Dockerfile` - Multi-stage Python build
4. `docker-compose.yml` - Multi-service orchestration
5. `.env.example` - Environment variable template
6. `devcontainer.json` - VS Code container integration

Project Structure

Directory Organization

```
~/projects/
│
```

```
├── web-apps/              # Web applications (Flask, FastAPI, Django)
│   ├── app-frontend/
│   └── app-backend/
│
├── microservices/         # Microservices architecture projects
│   ├── auth-service/
│   ├── payment-service/
│   └── notification-service/
│
├── ml-models/             # Machine learning projects
│   ├── image-classifier/
│   └── nlp-model/
│
├── databases/             # Standalone database containers
│   ├── postgres-main/
│   └── redis-cache/
│
├── mcp-services/          # MCP (Model Context Protocol) services
│   ├── mcp-filesystem/
│   ├── mcp-database/
│   └── mcp-api/
│
├── _templates/            # Project templates
│   └── python-microservice/
│
├── _shared/               # Shared resources
│   ├── docker-networks/   # Network configurations
│   ├── configs/           # Shared config files
│   ├── scripts/           # Helper scripts
│   └── volumes/           # Shared data volumes
│
└── _docs/                 # Documentation
    └── README.md
```

Daily Workflow

Creating a New Project

Method 1: Using Helper Script

bash

```
cd ~/projects/_shared/scripts

# Create project
./create-project.sh my-new-app microservices

# Open in VS Code
code ~/projects/microservices/my-new-app
```

Method 2: Manual Creation

```
bash

# Copy template
cp -r ~/projects/_templates/python-microservice ~/projects/microservices/my-app

# Navigate to project
cd ~/projects/microservices/my-app

# Initialize Git
git init
git add .
git commit -m "Initial commit"

# Create .env from template
cp .env.example .env
nano .env  # Edit with your values

# Open in VS Code
code .
```

Starting Development

```
bash
```

```
# Navigate to project
cd ~/projects/microservices/my-app

# Start containers
docker compose up -d

# View logs
docker compose logs -f

# Check running containers
docker compose ps

# Check resource usage
docker stats
```

Development in VS Code

1. **Open folder in WSL:**
   - `Ctrl+Shift+P` → "Remote-WSL: Open Folder in WSL"
   - Navigate to `~/projects/microservices/my-app`
2. **Reopen in Dev Container:**
   - `Ctrl+Shift+P` → "Dev Containers: Reopen in Container"
   - VS Code will build and connect to container
3. **Develop inside container:**
   - Terminal runs inside container
   - Extensions work inside container
   - No local venv needed!

Stopping Containers

```
bash
```

```
# Stop containers (keeps data)
docker compose down

# Stop and remove volumes (WARNING: deletes data)
docker compose down -v

# Stop all containers on system
docker stop $(docker ps -q)
```

Multi-Machine Git Workflow

Initial Setup on First Machine (Laptop)

```
bash

# Create project
cd ~/projects/microservices/my-app

# Initialize Git
git init
git add .
git commit -m "Initial project setup"

# Create GitHub repository (via web or CLI)
gh repo create my-app --private --source=. --remote=origin --push
```

Daily Workflow

Before Starting Work

```
bash

# ALWAYS pull first
git pull origin main

# Create feature branch
git checkout -b feature/my-feature
```

After Making Changes

```
bash
```

```
# Check status
git status

# Stage changes
git add .

# Commit with descriptive message
git commit -m "feat: Add user authentication"

# Push to remote
git push origin feature/my-feature
```

Switching Machines (Laptop ↔ Desktop)

On Current Machine (Before Switching)

```
bash

# Commit all work
git add .
git commit -m "WIP: Save progress before switching machines"

# Push to GitHub
git push origin feature/my-feature

# Stop containers (optional)
docker compose down
```

On New Machine (After Switching)

```
bash
```

```
# Clone repository (first time only)
git clone git@github.com:username/my-app.git ~/projects/microservices/my-app

# Or pull updates (if already cloned)
cd ~/projects/microservices/my-app
git pull origin feature/my-feature

# Copy .env from secure location or create new
cp .env.example .env
nano .env  # Update with machine-specific values

# Start containers
docker compose up -d

# Continue development
code .
```

Preventing Commit Issues

Common Problems:

1. **Forgot to push before switching:**
   - Work is lost or stuck on other machine
   - **Solution:** Always `git push` before shutdown
2. **Different .env files:**
   - Container won't start on new machine
   - **Solution:** Keep `.env.example` updated, use secrets manager for production
3. **Permission errors:**
   - Files created in container have wrong ownership
   - **Solution:** Always work in `~/projects`, not `/mnt/c`

Best Practices:

```
bash
```

```
# Pre-switch checklist script
cat > ~/projects/_shared/scripts/pre-switch.sh << 'EOF'
#!/bin/bash
echo "Pre-Switch Checklist:"
echo "1. Committing all changes..."
git add .
git commit -m "WIP: Save before machine switch" || echo "Nothing to commit"

echo "2. Pushing to remote..."
git push origin $(git branch --show-current)

echo "3. Stopping containers..."
docker compose down

echo "✓ Safe to switch machines!"
EOF

chmod +x ~/projects/_shared/scripts/pre-switch.sh
```

Resource Management

Monitoring Resource Usage

bash

```
# Real-time container stats
docker stats

# WSL2 memory usage (from PowerShell)
wsl -l -v
```

Optimizing Performance

1. Start Only What You Need

bash

```bash
# Start specific service
docker compose up -d web

# Start web + database (not cache)
docker compose up -d web db
```

2. Use Profiles for Optional Services

```bash
# Start without MCP services
docker compose up -d

# Start with MCP services
docker compose --profile mcp up -d
```

3. Clean Up Regularly

```bash
# Remove unused containers
docker container prune

# Remove unused images
docker image prune -a

# Remove unused volumes
docker volume prune

# Remove everything unused
docker system prune -a --volumes
```

4. Resource Limits Per Container

Edit `docker-compose.yml`:

```yaml
```

```yaml
services:
  web:
    deploy:
      resources:
        limits:
          cpus: '0.5'     # Max 50% of one CPU
          memory: 512M    # Max 512MB RAM
```

MCP Services Management

Understanding MCP Services

MCP (Model Context Protocol) services are specialized containers that:

- Connect to external APIs
- Access filesystems
- Query databases
- Often consume significant resources when idle

On-Demand MCP Strategy

1. Use Docker Compose Profiles

```yaml
# In docker-compose.yml
services:
  mcp-filesystem:
    # ... config ...
    profiles:
      - mcp
    restart: "no"  # Don't auto-restart
```

2. Start MCP Only When Needed

```bash
```

```
# Normal start (no MCP)
docker compose up -d

# Start with MCP
docker compose --profile mcp up -d mcp-filesystem

# Stop MCP when done
docker compose stop mcp-filesystem
```

3. Automated MCP Scripts

```
bash

# Create MCP toggle script
cat > ~/projects/_shared/scripts/toggle-mcp.sh << 'EOF'
#!/bin/bash
SERVICE=${1:-mcp-service}

if [ "$(docker compose ps -q $SERVICE)" ]; then
    echo "Stopping $SERVICE..."
    docker compose stop $SERVICE
else
    echo "Starting $SERVICE..."
    docker compose --profile mcp up -d $SERVICE
fi
EOF

chmod +x ~/projects/_shared/scripts/toggle-mcp.sh
```

Usage:

```
bash

# Toggle MCP on/off
~/projects/_shared/scripts/toggle-mcp.sh mcp-filesystem
```

Troubleshooting

Common Issues and Solutions

Issue 1: Docker Won't Start

Symptoms:

```
Cannot connect to the Docker daemon
```

Solution:

```bash
# Check if Docker is running
sudo systemctl status docker

# Start Docker
sudo systemctl start docker

# Enable auto-start
sudo systemctl enable docker
```

Issue 2: WSL2 Using Too Much Memory

Symptoms:

- Windows becomes slow
- `Vmmem` process uses 8GB+ RAM

Solution:

```powershell
# Reduce memory in .wslconfig
# Change to 4GB instead of 6GB
notepad ~/.wslconfig

# Restart WSL
wsl --shutdown
wsl
```

Issue 3: Slow File I/O

Symptoms:

- Docker builds take forever
- File operations are slow

Solution:

```bash
# Verify files are in WSL2 filesystem, not Windows
pwd
# Should show: /home/username/projects/...
# NOT: /mnt/c/Users/...

# Move project to WSL if needed
mv /mnt/c/Users/username/project ~/projects/
```

Issue 4: Git Permission Errors

Symptoms:

```
fatal: detected dubious ownership
```

Solution:

```bash
# Add directory to Git safe list
git config --global --add safe.directory ~/projects/microservices/my-app

# Fix file permissions
chmod -R u+rw ~/projects/microservices/my-app
```

Issue 5: Container Can't Connect to Database

Symptoms:

```
could not connect to server: Connection refused
```

Solution:

```bash
# Check if database is healthy
docker compose ps

# Check database logs
docker compose logs db

# Restart database
docker compose restart db

# Verify network
docker network ls
docker network inspect myapp_db-network
```

Issue 6: Port Already in Use

Symptoms:

```
bind: address already in use
```

Solution:

```bash
# Find process using port
sudo lsof -i :8000

# Kill process (replace PID)
kill -9 <PID>

# Or change port in docker-compose.yml
ports:
  - "8001:8000"  # Use different host port
```

Best Practices

1. File Organization

✅ **DO:**

- Store ALL code in `~/projects` (WSL2 filesystem)
- Use separate repositories per project
- Keep `.env.example` updated
- Commit `.gitignore`, `.dockerignore`, `docker-compose.yml`

❌ **DON'T:**

- Store code in `/mnt/c/Users/` (Windows filesystem)
- Mix unrelated projects in one repository
- Commit `.env` files (secrets)
- Commit `node_modules/`, `__pycache__/`, `venv/`

2. Docker Best Practices

✅ **DO:**

- Use multi-stage builds (smaller images)
- Set resource limits for each container
- Use health checks
- Clean up unused containers/images regularly
- Use `.dockerignore` to exclude unnecessary files

❌ **DON'T:**

- Run containers as root user
- Store data inside containers (use volumes)
- Expose all ports publicly
- Run containers without resource limits

3. Git Workflow

✅ **DO:**

- Pull before starting work
- Commit frequently with meaningful messages
- Push before switching machines

- Use feature branches
- Keep commits atomic (one logical change)

❌ **DON'T:**

- Commit sensitive data (API keys, passwords)
- Make huge commits with multiple unrelated changes
- Force push to main/master
- Work directly on main branch

4. VS Code Development

✅ **DO:**

- Use Dev Containers for consistent environments
- Configure formatters (Black, Prettier)
- Use linters (Pylint, Flake8)
- Enable auto-save
- Use integrated terminal

❌ **DON'T:**

- Open projects from Windows side
- Ignore linter warnings
- Commit with formatting errors

5. Resource Management

✅ **DO:**

- Monitor `docker stats` regularly
- Stop unused containers
- Use profiles for optional services
- Set memory/CPU limits
- Clean up weekly

❌ **DON'T:**

- Run all containers simultaneously
- Ignore high memory usage
- Let Docker Desktop auto-start
- Keep old images indefinitely

Appendix: Quick Reference

Essential Commands

bash

```
# WSL Management
wsl --shutdown          # Restart WSL2
wsl --status            # Check WSL version
wsl -l -v               # List distributions

# Docker
docker ps               # List running containers
docker ps -a            # List all containers
docker stats            # Resource usage
docker system df        # Disk usage
docker system prune -a  # Clean everything

# Docker Compose
docker compose up -d    # Start services
docker compose down     # Stop services
docker compose ps       # List services
docker compose logs -f  # Follow logs
docker compose restart  # Restart services

# Git
git status              # Check status
git add .               # Stage all changes
git commit -m "message" # Commit
git push origin branch  # Push to remote
git pull origin branch  # Pull from remote

# Project Management
code ~/projects         # Open in VS Code
tree ~/projects -L 2    # View structure
```

File Locations Reference

File	Location	Purpose
`.wslconfig`	`C:\Users\<User>\.wslconfig`	WSL2 global settings (Windows)
`wsl.conf`	`/etc/wsl.conf`	WSL2 distribution settings (Linux)
`daemon.json`	`%USERPROFILE%\.docker\daemon.json`	Docker daemon config
Projects	`~/projects/` or `/home/username/projects/`	All code (WSL2)
Templates	`~/projects/_templates/`	Project templates

File	Location	Purpose
Scripts	`~/projects/_shared/scripts/`	Helper scripts

Summary

What We Built

1. ✅ **WSL2 with optimized resource limits** (6GB RAM, 3 CPUs)
2. ✅ **Docker Desktop with WSL2 backend** (4GB RAM, 2 CPUs)
3. ✅ **Professional project structure** in WSL2 filesystem
4. ✅ **Containerized development** (no local venvs needed)
5. ✅ **VS Code integration** with Dev Containers
6. ✅ **Multi-machine Git workflow** (laptop ↔ desktop)
7. ✅ **On-demand MCP services** (resource-efficient)
8. ✅ **Production-grade templates** (.gitignore, Dockerfile, docker-compose.yml)

Performance Benefits

- **2-5x faster I/O** using WSL2 filesystem
- **50% memory savings** with resource limits
- **Zero dependency conflicts** with containers
- **Identical environments** across all machines

Next Steps for Students

1. Complete the installation following this guide
2. Create your first project from template
3. Practice the Git workflow
4. Experiment with different project types
5. Monitor and optimize resource usage
6. Share your setup with classmates

🎉 **Congratulations! You now have a production-grade development environment.**

For questions or issues, refer to the Troubleshooting section or create an issue in the course repository.