npm start

```""" % name)

if has\_rich:

Console().print(f"[bold {current\_theme['success']}]✅ Project {name} created successfully![/]")

Console().print(f"Location: {project\_dir}")

else:

print(colored(f"✅ Project {name} created successfully!", current\_theme["success"]))

print(f"Location: {project\_dir}")

@project.command("list")

def project\_list():

"""List all projects"""

projects\_dir = os.path.join(BASE\_DIR, "projects")

if not os.path.exists(projects\_dir):

print("No projects found.")

return

projects = [d for d in os.listdir(projects\_dir)

if os.path.isdir(os.path.join(projects\_dir, d))]

if not projects:

print("No projects found.")

return

if has\_rich:

console = Console()

table = Table(show\_header=True, header\_style=f"bold {current\_theme['accent']}")

table.add\_column("Project Name")

table.add\_column("Type")

table.add\_column("Last Modified")

for project in projects:

project\_path = os.path.join(projects\_dir, project)

# Determine project type

project\_type = "Unknown"

if os.path.exists(os.path.join(project\_path, "package.json")):

project\_type = "Node.js"

elif os.path.exists(os.path.join(project\_path, "requirements.txt")):

project\_type = "Python"

elif os.path.exists(os.path.join(project\_path, "index.html")):

project\_type = "Web"

# Get last modified time

try:

mtime = os.path.getmtime(project\_path)

last\_modified = datetime.fromtimestamp(mtime).strftime("%Y-%m-%d %H:%M")

except:

last\_modified = "Unknown"

table.add\_row(project, project\_type, last\_modified)

console.print("\n[bold]Projects:[/bold]")

console.print(table)

else:

print(colored("Projects:", current\_theme["success"], attrs=["bold"]))

for project in projects:

print(f" - {project}")

@cli.group()

def deploy():

"""Deployment commands"""

pass

@deploy.command("vercel")

@click.argument("project")

def deploy\_vercel(project):

"""Deploy a project to Vercel"""

project\_dir = os.path.join(BASE\_DIR, "projects", project)

if not os.path.exists(project\_dir):

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error:[/] Project {project} not found")

else:

print(colored(f"Error: Project {project} not found", current\_theme["error"]))

return

if has\_rich:

Console().print(f"[bold {current\_theme['info']}]Deploying {project} to Vercel...[/]")

Console().print("[bold]Checking for Vercel CLI...[/]")

else:

print(colored(f"Deploying {project} to Vercel...", current\_theme["info"]))

print("Checking for Vercel CLI...")

# Check if vercel CLI is installed

try:

subprocess.run(["vercel", "--version"], capture\_output=True, check=True)

except (subprocess.CalledProcessError, FileNotFoundError):

if has\_rich:

Console().print("[yellow]Vercel CLI not found. Installing...[/]")

else:

print("Vercel CLI not found. Installing...")

try:

subprocess.run(["npm", "install", "-g", "vercel"], check=True)

except subprocess.CalledProcessError:

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error installing Vercel CLI[/]")

else:

print(colored("Error installing Vercel CLI", current\_theme["error"]))

return

# Deploy using Vercel

try:

os.chdir(project\_dir)

subprocess.run(["vercel", "--prod"], check=True)

except subprocess.CalledProcessError:

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Deployment failed[/]")

else:

print(colored("Deployment failed", current\_theme["error"]))

return

if has\_rich:

Console().print(f"[bold {current\_theme['success']}]✅ Deployment successful![/]")

else:

print(colored("✅ Deployment successful!", current\_theme["success"]))

@deploy.command("render")

@click.argument("project")

def deploy\_render(project):

"""Deploy a project to Render"""

project\_dir = os.path.join(BASE\_DIR, "projects", project)

if not os.path.exists(project\_dir):

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error:[/] Project {project} not found")

else:

print(colored(f"Error: Project {project} not found", current\_theme["error"]))

return

if has\_rich:

Console().print(f"[bold {current\_theme['info']}]Deploying {project} to Render...[/]")

Console().print("[yellow]Note: This requires the Render CLI or API key setup.[/]")

else:

print(colored(f"Deploying {project} to Render...", current\_theme["info"]))

print("Note: This requires the Render CLI or API key setup.")

# Check for render.yaml

render\_config = os.path.join(project\_dir, "render.yaml")

if not os.path.exists(render\_config):

if has\_rich:

Console().print("[yellow]render.yaml not found. Creating a template...[/]")

else:

print("render.yaml not found. Creating a template...")

with open(render\_config, "w") as f:

f.write("""services:

- type: web

name: %s

env: auto

buildCommand: npm install

startCommand: npm start

envVars:

- key: NODE\_ENV

value: production

""" % project)

if has\_rich:

Console().print(f"[bold {current\_theme['warning']}]Created render.yaml template. Please edit it for your specific needs.[/]")

Console().print("Then run this command again to deploy.")

else:

print(colored("Created render.yaml template. Please edit it for your specific needs.", current\_theme["warning"]))

print("Then run this command again to deploy.")

return

if has\_rich:

console = Console()

console.print("\n[bold]To complete deployment:[/]")

console.print("1. Visit [link]https://render.com/deploy[/link]")

console.print("2. Connect your GitHub repository")

console.print(f"3. Select the {project} repository")

console.print("4. Follow the prompts to deploy")

console.print(f"\n[bold {current\_theme['success']}]Deployment instructions provided[/]")

else:

print("To complete deployment:")

print("1. Visit https://render.com/deploy")

print("2. Connect your GitHub repository")

print(f"3. Select the {project} repository")

print("4. Follow the prompts to deploy")

print(colored("Deployment instructions provided", current\_theme["success"]))

@deploy.command("cloudflare")

@click.argument("project")

def deploy\_cloudflare(project):

"""Deploy a project to Cloudflare Pages"""

project\_dir = os.path.join(BASE\_DIR, "projects", project)

if not os.path.exists(project\_dir):

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error:[/] Project {project} not found")

else:

print(colored(f"Error: Project {project} not found", current\_theme["error"]))

return

if has\_rich:

Console().print(f"[bold {current\_theme['info']}]Deploying {project} to Cloudflare Pages...[/]")

else:

print(colored(f"Deploying {project} to Cloudflare Pages...", current\_theme["info"]))

# Check if Cloudflare CLI (Wrangler) is installed

try:

subprocess.run(["wrangler", "--version"], capture\_output=True, check=True)

except (subprocess.CalledProcessError, FileNotFoundError):

if has\_rich:

Console().print("[yellow]Cloudflare Wrangler CLI not found. Installing...[/]")

else:

print("Cloudflare Wrangler CLI not found. Installing...")

try:

subprocess.run(["npm", "install", "-g", "wrangler"], check=True)

except subprocess.CalledProcessError:

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error installing Wrangler CLI[/]")

else:

print(colored("Error installing Wrangler CLI", current\_theme["error"]))

return

# Create wrangler.toml if doesn't exist

wrangler\_config = os.path.join(project\_dir, "wrangler.toml")

if not os.path.exists(wrangler\_config):

if has\_rich:

Console().print("[yellow]wrangler.toml not found. Creating a template...[/]")

else:

print("wrangler.toml not found. Creating a template...")

with open(wrangler\_config, "w") as f:

f.write("""name = "%s"

type = "webpack"

account\_id = ""

workers\_dev = true

route = ""

zone\_id = ""

[site]

bucket = "."

entry-point = "."

""" % project)

if has\_rich:

Console().print(f"[bold {current\_theme['warning']}]Created wrangler.toml template. Please edit it with your Cloudflare account details.[/]")

Console().print("Then run this command again to deploy.")

else:

print(colored("Created wrangler.toml template. Please edit it with your Cloudflare account details.", current\_theme["warning"]))

print("Then run this command again to deploy.")

return

# Deploy using Wrangler

try:

os.chdir(project\_dir)

subprocess.run(["wrangler", "publish"], check=True)

except subprocess.CalledProcessError:

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Deployment failed[/]")

else:

print(colored("Deployment failed", current\_theme["error"]))

return

if has\_rich:

Console().print(f"[bold {current\_theme['success']}]✅ Deployment successful![/]")

else:

print(colored("✅ Deployment successful!", current\_theme["success"]))

@cli.group()

def api():

"""API tools"""

pass

@api.command("generate")

@click.argument("name")

@click.option("--type", "-t", default="rest", help="API type (rest, graphql)")

def api\_generate(name, type):

"""Generate API boilerplate"""

api\_dir = os.path.join(BASE\_DIR, "api", name)

if os.path.exists(api\_dir):

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error:[/] API {name} already exists")

else:

print(colored(f"Error: API {name} already exists", current\_theme["error"]))

return

if has\_rich:

Console().print(f"[bold {current\_theme['info']}]Generating {type} API:[/] {name}...")

else:

print(colored(f"Generating {type} API: {name}...", current\_theme["info"]))

# Create API directory

os.makedirs(api\_dir, exist\_ok=True)

if type == "rest":

# Create basic REST API structure

os.makedirs(os.path.join(api\_dir, "src"), exist\_ok=True)

os.makedirs(os.path.join(api\_dir, "src", "routes"), exist\_ok=True)

os.makedirs(os.path.join(api\_dir, "src", "controllers"), exist\_ok=True)

os.makedirs(os.path.join(api\_dir, "src", "models"), exist\_ok=True)

# Create package.json

with open(os.path.join(api\_dir, "package.json"), "w") as f:

f.write("""{

"name": "%s-api",

"version": "1.0.0",

"description": "REST API",

"main": "src/index.js",

"scripts": {

"start": "node src/index.js",

"dev": "nodemon src/index.js"

},

"dependencies": {

"express": "^4.17.1",

"cors": "^2.8.5",

"dotenv": "^10.0.0",

"mongoose": "^6.0.8"

},

"devDependencies": {

"nodemon": "^2.0.12"

}

}""" % name)

# Create index.js

with open(os.path.join(api\_dir, "src", "index.js"), "w") as f:

f.write("""const express = require('express');

const cors = require('cors');

require('dotenv').config();

const app = express();

const port = process.env.PORT || 3000;

app.use(cors());

app.use(express.json());

// Routes

app.get('/', (req, res) => {

res.json({ message: 'Welcome to %s API' });

});

// Example route

app.use('/api/items', require('./routes/items'));

app.listen(port, () => {

console.log(`Server running on port ${port}`);

});""" % name)

# Create example route

with open(os.path.join(api\_dir, "src", "routes", "items.js"), "w") as f:

f.write("""const express = require('express');

const router = express.Router();

const ItemController = require('../controllers/item.controller');

// Get all items

router.get('/', ItemController.getAll);

// Get item by ID

router.get('/:id', ItemController.getById);

// Create new item

router.post('/', ItemController.create);

// Update item

router.put('/:id', ItemController.update);

// Delete item

router.delete('/:id', ItemController.delete);

module.exports = router;""")

# Create example controller

with open(os.path.join(api\_dir, "src", "controllers", "item.controller.js"), "w") as f:

f.write("""// Example item controller

// In a real app, this would interact with a database

let items = [

{ id: 1, name: 'Item 1', description: 'This is item 1' },

{ id: 2, name: 'Item 2', description: 'This is item 2' }

];

exports.getAll = (req, res) => {

res.json(items);

};

exports.getById = (req, res) => {

const item = items.find(i => i.id === parseInt(req.params.id));

if (!item) return res.status(404).json({ message: 'Item not found' });

res.json(item);

};

exports.create = (req, res) => {

const { name, description } = req.body;

if (!name) return res.status(400).json({ message: 'Name is required' });

const newItem = {

id: items.length + 1,

name,

description

};

items.push(newItem);

res.status(201).json(newItem);

};

exports.update = (req, res) => {

const item = items.find(i => i.id === parseInt(req.params.id));

if (!item) return res.status(404).json({ message: 'Item not found' });

const { name, description } = req.body;

if (name) item.name = name;

if (description) item.description = description;

res.json(item);

};

exports.delete = (req, res) => {

const itemIndex = items.findIndex(i => i.id === parseInt(req.params.id));

if (itemIndex === -1) return res.status(404).json({ message: 'Item not found' });

items.splice(itemIndex, 1);

res.status(204).end();

};""")

# Create .env

with open(os.path.join(api\_dir, ".env"), "w") as f:

f.write("""PORT=3000""")

elif type == "graphql":

# Create basic GraphQL API structure

os.makedirs(os.path.join(api\_dir, "src"), exist\_ok=True)

os.makedirs(os.path.join(api\_dir, "src", "schema"), exist\_ok=True)

os.makedirs(os.path.join(api\_dir, "src", "resolvers"), exist\_ok=True)

# Create package.json

with open(os.path.join(api\_dir, "package.json"), "w") as f:

f.write("""{

"name": "%s-graphql-api",

"version": "1.0.0",

"description": "GraphQL API",

"main": "src/index.js",

"scripts": {

"start": "node src/index.js",

"dev": "nodemon src/index.js"

},

"dependencies": {

"apollo-server-express": "^3.5.0",

"express": "^4.17.1",

"graphql": "^16.0.0",

"dotenv": "^10.0.0"

},

"devDependencies": {

"nodemon": "^2.0.12"

}

}""" % name)

# Create index.js

with open(os.path.join(api\_dir, "src", "index.js"), "w") as f:

f.write("""const express = require('express');

const { ApolloServer } = require('apollo-server-express');

require('dotenv').config();

const typeDefs = require('./schema');

const resolvers = require('./resolvers');

async function startServer() {

const app = express();

const port = process.env.PORT || 4000;

const server = new ApolloServer({

typeDefs,

resolvers,

});

await server.start();

server.applyMiddleware({ app });

app.listen(port, () => {

console.log(`GraphQL server running at http://localhost:${port}${server.graphqlPath}`);

});

}

startServer();""")

# Create schema

with open(os.path.join(api\_dir, "src", "schema", "index.js"), "w") as f:

f.write("""const { gql } = require('apollo-server-express');

const typeDefs = gql`

type Item {

id: ID!

name: String!

description: String

}

type Query {

items: [Item]

item(id: ID!): Item

}

type Mutation {

createItem(name: String!, description: String): Item

updateItem(id: ID!, name: String, description: String): Item

deleteItem(id: ID!): Boolean

}

`;

module.exports = typeDefs;""")

# Create resolvers

with open(os.path.join(api\_dir, "src", "resolvers", "index.js"), "w") as f:

f.write("""// Example in-memory data store

let items = [

{ id: '1', name: 'Item 1', description: 'This is item 1' },

{ id: '2', name: 'Item 2', description: 'This is item 2' },

];

const resolvers = {

Query: {

items: () => items,

item: (\_, { id }) => items.find(item => item.id === id),

},

Mutation: {

createItem: (\_, { name, description }) => {

const id = String(items.length + 1);

const newItem = { id, name, description };

items.push(newItem);

return newItem;

},

updateItem: (\_, { id, name, description }) => {

const itemIndex = items.findIndex(item => item.id === id);

if (itemIndex === -1) return null;

const updatedItem = {

...items[itemIndex],

name: name || items[itemIndex].name,

description: description !== undefined ? description : items[itemIndex].description,

};

items[itemIndex] = updatedItem;

return updatedItem;

},

deleteItem: (\_, { id }) => {

const itemIndex = items.findIndex(item => item.id === id);

if (itemIndex === -1) return false;

items.splice(itemIndex, 1);

return true;

},

},

};

module.exports = resolvers;""")

# Create .env

with open(os.path.join(api\_dir, ".env"), "w") as f:

f.write("""PORT=4000""")

if has\_rich:

Console().print(f"[bold {current\_theme['success']}]✅ API {name} generated successfully![/]")

Console().print(f"Location: {api\_dir}")

Console().print(f"Install dependencies: cd {api\_dir} && npm install")

Console().print(f"Start the API: npm run dev")

else:

print(colored(f"✅ API {name} generated successfully!", current\_theme["success"]))

print(f"Location: {api\_dir}")

print(f"Install dependencies: cd {api\_dir} && npm install")

print(f"Start the API: npm run dev")

@cli.group()

def github():

"""GitHub integration commands"""

pass

@github.command("setup")

@click.argument("project")

def github\_setup(project):

"""Set up GitHub repository for a project"""

project\_dir = os.path.join(BASE\_DIR, "projects", project)

if not os.path.exists(project\_dir):

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error:[/] Project {project} not found")

else:

print(colored(f"Error: Project {project} not found", current\_theme["error"]))

return

if has\_rich:

Console().print(f"[bold {current\_theme['info']}]Setting up GitHub repository for {project}...[/]")

else:

print(colored(f"Setting up GitHub repository for {project}...", current\_theme["info"]))

# Check if git is already initialized

git\_dir = os.path.join(project\_dir, ".git")

if os.path.exists(git\_dir):

if has\_rich:

Console().print("Git repository already initialized.")

else:

print("Git repository already initialized.")

else:

if has\_rich:

Console().print("Initializing git repository...")

else:

print("Initializing git repository...")

try:

subprocess.run(["git", "init"], cwd=project\_dir, check=True)

except subprocess.CalledProcessError:

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Failed to initialize git repository[/]")

else:

print(colored("Failed to initialize git repository", current\_theme["error"]))

return

# Create .gitignore if doesn't exist

gitignore\_file = os.path.join(project\_dir, ".gitignore")

if not os.path.exists(gitignore\_file):

with open(gitignore\_file, "w") as f:

f.write("""node\_modules/

.env

.env.local

.DS\_Store

\*.log

dist/

build/

""")

# Set up GitHub CLI if available

try:

subprocess.run(["gh", "--version"], capture\_output=True, check=True)

create\_repo = input("Create GitHub repository now? (y/n): ")

if create\_repo.lower() == "y":

try:

# Create GitHub repository

if has\_rich:

Console().print("Creating GitHub repository...")

else:

print("Creating GitHub repository...")

subprocess.run(["gh", "repo", "create", project, "--source=.", "--push"], cwd=project\_dir, check=True)

if has\_rich:

Console().print(f"[bold {current\_theme['success']}]✅ GitHub repository created and code pushed![/]")

else:

print(colored("✅ GitHub repository created and code pushed!", current\_theme["success"]))

except subprocess.CalledProcessError:

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Failed to create GitHub repository[/]")

Console().print("Make sure you're logged in with 'gh auth login'")

else:

print(colored("Failed to create GitHub repository", current\_theme["error"]))

print("Make sure you're logged in with 'gh auth login'")

except (subprocess.CalledProcessError, FileNotFoundError):

if has\_rich:

Console().print("[yellow]GitHub CLI not found.[/]")

Console().print("To create a repository manually:")

Console().print("1. Visit [link]https://github.com/new[/link]")

Console().print("2. Create a repository named", project)

Console().print(f"3. Run the following commands in {project\_dir}:")

Console().print(" git remote add origin https://github.com/YOUR-USERNAME/" + project + ".git")

Console().print(" git branch -M main")

Console().print(" git push -u origin main")

else:

print("GitHub CLI not found.")

print("To create a repository manually:")

print("1. Visit https://github.com/new")

print("2. Create a repository named", project)

print(f"3. Run the following commands in {project\_dir}:")

print(" git remote add origin https://github.com/YOUR-USERNAME/" + project + ".git")

print(" git branch -M main")

print(" git push -u origin main")

if has\_rich:

Console().print(f"[bold {current\_theme['success']}]Git setup complete![/]")

else:

print(colored("Git setup complete!", current\_theme["success"]))

@cli.group()

def tunnel():

"""SSH tunneling commands"""

pass

@tunnel.command("create")

@click.argument("port", type=int)

@click.option("--remote-port", "-r", type=int, help="Remote port (defaults to same as local)")

def tunnel\_create(port, remote\_port):

"""Create an SSH tunnel for a local port"""

if not remote\_port:

remote\_port = port

if has\_rich:

Console().print(f"[bold {current\_theme['info']}]Setting up SSH tunnel from local port {port} to remote port {remote\_port}...[/]")

Console().print("[yellow]Note: This requires SSH access to a remote server.[/]")

else:

print(colored(f"Setting up SSH tunnel from local port {port} to remote port {remote\_port}...", current\_theme["info"]))

print("Note: This requires SSH access to a remote server.")

# Get SSH connection details

ssh\_host = input("Enter SSH host (e.g., user@example.com): ")

if not ssh\_host:

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error: SSH host is required[/]")

else:

print(colored("Error: SSH host is required", current\_theme["error"]))

return

try:

# Create tunnel using SSH

if has\_rich:

Console().print(f"Creating tunnel {port} -> {ssh\_host}:{remote\_port}")

else:

print(f"Creating tunnel {port} -> {ssh\_host}:{remote\_port}")

# Use -N flag to not execute a remote command, just forward ports

subprocess.Popen(["ssh", "-N", "-L", f"{port}:localhost:{remote\_port}", ssh\_host])

if has\_rich:

Console().print(f"[bold {current\_theme['success']}]✅ Tunnel established! Local port {port} is now forwarded to {ssh\_host}:{remote\_port}[/]")

Console().print("The tunnel will remain active until this process is terminated.")

Console().print("[italic]Press Ctrl+C to stop the tunnel.[/]")

else:

print(colored(f"✅ Tunnel established! Local port {port} is now forwarded to {ssh\_host}:{remote\_port}", current\_theme["success"]))

print("The tunnel will remain active until this process is terminated.")

print("Press Ctrl+C to stop the tunnel.")

except Exception as e:

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error creating tunnel: {str(e)}[/]")

else:

print(colored(f"Error creating tunnel: {str(e)}", current\_theme["error"]))

@cli.command()

def config():

"""Configure Triad Terminal settings"""

print\_header()

if has\_rich:

console = Console()

console.print("\n[bold]Triad Terminal Configuration[/]\n")

# Current settings table

table = Table(title="Current Settings", show\_header=True, header\_style=f"bold {current\_theme['accent']}")

table.add\_column("Setting")

table.add\_column("Value")

for key, value in config.items():

if key == "api\_keys":

table.add\_row(key, f"{len(value)} keys configured")

elif isinstance(value, dict):

table.add\_row(key, str(value))

else:

table.add\_row(key, str(value))

console.print(table)

console.print("\nAvailable themes: [green]matrix[/], [blue]cyberpunk[/], [magenta]synthwave[/], [red]bloodmoon[/]")

else:

print("\nTriad Terminal Configuration\n")

print("Current settings:")

for key, value in config.items():

if key == "api\_keys":

print(f" {key}: {len(value)} keys configured")

elif isinstance(value, dict):

print(f" {key}:")

for k, v in value.items():

print(f" {k}: {v}")

else:

print(f" {key}: {value}")

print("\nAvailable themes: matrix, cyberpunk, synthwave, bloodmoon")

theme = input(f"\nSelect theme [{config.get('theme', 'matrix')}]: ")

if theme:

config["theme"] = theme

user\_name = input(f"\nUser name [{config.get('user', {}).get('name', '')}]: ")

if user\_name:

if "user" not in config:

config["user"] = {}

config["user"]["name"] = user\_name

user\_email = input(f"\nUser email [{config.get('user', {}).get('email', '')}]: ")

if user\_email:

if "user" not in config:

config["user"] = {}

config["user"]["email"] = user\_email

# Save config

with open(config\_file, "w") as f:

yaml.dump(config, f)

if has\_rich:

Console().print(f"\n[bold {current\_theme['success']}]✅ Configuration saved![/]")

else:

print(colored("\n✅ Configuration saved!", current\_theme["success"]))

print\_footer()

# Show matrix animation if theme is matrix

if config.get("theme") == "matrix":

show\_matrix\_animation(3)

if \_\_name\_\_ == "\_\_main\_\_":

if len(sys.argv) == 1:

# If no arguments, run the start command

sys.argv.append("start")

try:

cli()

except Exception as e:

if has\_rich:

Console().print(f"[bold {current\_theme['error']}]Error: {str(e)}[/]")

else:

print(colored(f"Error: {str(e)}", current\_theme["error"]))

sys.exit(1)

EOF

# Make script executable

chmod +x "$BASE\_DIR/bin/triad"

echo -e "\e[38;5;46m✅ Terminal script created!\e[0m"

return 0

}

# Set up environment for terminal

setup\_environment() {

echo -e "\e[38;5;46m🛠️ Setting up environment...\e[0m"

# Create bash/zsh integration script

cat > "$CONFIG\_DIR/shell\_integration.sh" << 'EOF'

#!/bin/bash

# Triad Terminal Shell Integration

# Add Triad bin directory to PATH

export PATH="$HOME/.triad/bin:$PATH"

# Aliases

alias tt="triad"

alias ttp="triad project"

alias ttd="triad deploy"

alias tta="triad api"

alias ttg="triad github"

alias tts="triad start"

# Terminal welcome message

if [[ "$TERM\_PROGRAM" != "vscode" ]]; then

# Get a random color each time

colors=("\033[38;5;39m" "\033[38;5;46m" "\033[38;5;213m" "\033[38;5;196m")

color=${colors[$RANDOM % ${#colors[@]}]}

echo -e "${color}"

echo "╔════════════════════════════════════════════╗"

echo "║ Welcome to Triad Terminal ║"

echo "║ ║"

echo "║ Type 'tt' or 'triad' to start ║"

echo "╚════════════════════════════════════════════╝"

echo -e "\033[0m"

# Show matrix animation if available and randomly (33% chance)

if [[ -f "$HOME/.triad/ascii\_art/matrix.