Analysis Grand Challenge Related Tests

Prompt

> Implement a complete H->ZZ->4l analysis pipeline in the template file `analysis.py`. Make sure `analysis.py` is PEP 723 compliant, and use the `uv` tools to run and test it. Use the appropriate `agc-skills` skills for further instructions.
>
> Create a new code block (separated by an empty line) for each major step. The pipeline consists of four main stages, to be implemented in order:
>
> Dataset retrieval - Retrieve the input columns from the input_files dictionary (Data, Background Z ttbar, Background ZZ*, Signal). Tree name: mini. Event selection: exactly four leptons (lep_n == 4). Columns needed downstream: lepton pt, eta, phi, energy, charge, type, mcWeight, and the scale factors scaleFactor_ELE, scaleFactor_MUON, scaleFactor_LepTRIGGER, scaleFactor_PILEUP. Compute a scale factor variation systematic at query stage (product of the four scale factors above, scaled up by 1.1 and down by 0.9). Support both a func_adl (FuncADL_Uproot) query and an UprootRaw alternative, selectable via a USE_SERVICEX_UPROOT_RAW flag, and respect an IGNORE_CACHE flag.
>
> Event processing - Process the retrieved columns to build the discriminating variable m4l (invariant mass of the four-lepton system). Select events with exactly two same-flavor opposite-charge lepton pairs (electrons or muons). Produce histograms of m4l for one signal region, for each sample (Data, Signal MC, Background ZZ*, Background Z ttbar), including variations for the following systematics: scale factor variation, 4l mass variation, and ZZ normalization. Also account for a signal normalization factor.
>
> Plotting - Plot the m4l histograms for the signal region, showing all samples and the systematic variations listed above. Create a figures directory as a sub-directory of the working sandbox directory if one does not already exist, and save each plot as both a .pdf and a .png file in that directory.
>
> Statistical analysis - Using the histograms produced above, build a cabinetry configuration, collect and postprocess the templates, build a workspace, and perform the fit. The parameter of interest is the signal normalization. Produce and save (as .pdf and .png in the figures

> directory) a pull plot (name: pull_plot.pdf/.png), a correlation matrix plot (name: matrix.pdf/.png), and a post-fit data/model comparison plot (name: model-comp.pdf/.png) for the signal region with a meaningful x-axis label. Finally, write a results.json file containing the POI value and error, formatted as:
>
> {
>
> “poi_val” : x,
>
> “poi_err” : y
>
> }
>
> After each stage, run the Analysis.py file and capture any error output. If there is an error, examine the message and correct it to the best of your ability, then re-run until the file executes without errors and the output looks as expected.

Final response

> Implemented the complete PEP 723 pipeline in `analysis.py`.
>
> - All four stages ran successfully with `uv`.
> - Both query backends produced identical results.
> - Five tests passed.
> - PDF/PNG plots saved in `figures/`; stage logs saved in `logs/`.
> - `results.json`: **POI = 1.136738 ± 0.702181**.
>
> Run again with:
>
> ```
> uv run --locked analysis.py
> ```

Skills and marketplaces

- Marketplaces:
  - `https://github.com/gordonwatts/agc_marketplace`
- Plugins:

  - `agc-skills@agc_marketplace`
- Skill directories:
  - `plugins/cache/agc_marketplace/agc-skills/0.1.0/skills/cabinetry-analysis`
  - `plugins/cache/agc_marketplace/agc-skills/0.1.0/skills/coffea-processing`
  - `plugins/cache/agc_marketplace/agc-skills/0.1.0/skills/corrections-systematics`
  - `plugins/cache/agc_marketplace/agc-skills/0.1.0/skills/dataset-retrieval`
  - `plugins/cache/agc_marketplace/agc-skills/0.1.0/skills/hist-plotter`

Run summary

Field	Value
Run date	2026-09-07
Started	02:56:14
Finished	03:05:22
Total duration	0h 9m 8s
Codex execution	0h 8m 57s
Status	succeeded
Exit code	0
Distribution	Ubuntu
Codex version	codex-cli 0.153.4
Workspace	/home/gwatts/llm-testing/test-wsl2-llm-pauOxkCJ/workspace
Workspace retained	True

Phase timing

Phase	Started	Finished	Duration
preflight	02:56:14	02:56:16	2s
workspace_creation	02:56:16	02:56:16	0s
input_transfer	02:56:16	02:56:19	3s
codex_home_setup	02:56:19	02:56:19	0s
plugin_installation	02:56:19	02:56:20	1s

Phase	Started	Finished	Duration
codex_execution	02:56:20	03:05:17	8m 57s
workspace_inventory	03:05:17	03:05:18	0s
copy_back	03:05:18	03:05:21	3s
session_trace_collection	03:05:21	03:05:21	0s

Token usage and cost

- Pricing file: `bundled:model-pricing.yaml`
- Currency: `USD`
- Token counts come from the `usage` object on each captured `turn.completed` event in Codex stdout JSONL; the configured model is used because those events do not report a model name.
- `input_tokens` is the total input-token count, including the `cached_input_tokens` subset. Uncached input is calculated as `input_tokens - cached_input_tokens`; cached input was served from the prompt cache and is priced at its separate cached-input rate.
- Output tokens are reported separately. Costs use the uncached-input rate for uncached input, the cached-input rate for cached input, and the output rate for output tokens.

Model	Attribution	Input	Cached input	Output	Reasoning output	USD total
gpt-6-astra	inferred - model not directly reported	1,424,424	1,358,720	13,205	1,260	$2.68
Aggregate						**$2.68**

Invocation

> C:\Users\gordo\AppData\Local\uv\cache\archive-v0\ORp9JIYJhVbwSU8c\Scripts\test-wsl2-llm template run .\agc-tests.yaml --model gpt-6-astra:medium

Codex command

```
# TEST_WSL2_LLM_ARG_* values are base64-encoded WSL transport arguments (such as paths ar
wsl.exe -d Ubuntu -- env TEST_WSL2_LLM_ARG_0=L2hvbWUvZ3dhdHRzL2xsbS10ZXN0aW5nL3Rlc3Qtd3N
```

▶ Workspace inventory

Copied-back files

agc-q1-full.analysis.py

- Source: `analysis.py`
- Size: 25,183 bytes

Text contents (first 10 lines visible; scroll for the rest):

```
# /// script
# requires-python = ">=3.11,<3.13"
# dependencies = [
#   "awkward>=2.8,<3", "cabinetry>=0.6,<0.7", "coffea>=2026.7,<2027",
#   "hist>=2.8,<3", "matplotlib>=3.9,<4", "mplhep>=0.3,<1",
#   "numpy>=1.26,<3", "pyhf>=0.7,<0.8", "pyyaml>=6,<7",
#   "servicex>=3,<4", "func-adl>=3.2,<4", "uproot>=5.6,<6",
#   "vector>=1.6,<2", "iminuit>=2.30,<3",
# ]
# ///

# ---
# jupyter:
#   jupytext:
#     formats: ipynb,py:percent
#     text_representation:
#       extension: .py
#       format_name: percent
#       format_version: '1.3'
#       jupytext_version: 1.19.1
#   kernelspec:
#     display_name: Python 3 (ipykernel)
#     language: python
#     name: python3
# ---

# %% [markdown]
# # ATLAS Open Data $H\rightarrow ZZ^\star$ with `ServiceX`, `coffea`, `cabinetry` & `pyl

# %%
import argparse
import json
import logging
import os
import re
import time
from pathlib import Path

os.environ.setdefault("MPLCONFIGDIR", str(Path.cwd() / ".cache" / "matplotlib"))

import awkward as ak
import cabinetry
import hist
import matplotlib
import mplhep
```

```
import numpy as np
import uproot
from coffea import processor
from coffea.nanoevents import BaseSchema, NanoEventsFactory

matplotlib.use("Agg")
import matplotlib.pyplot as plt
import servicex
import vector
import yaml

vector.register_awkward()

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("hzz")

# %%
# set some global settings

# chunk size to use
CHUNKSIZE = 500_000

# ServiceX behavior: ignore cache with repeated queries
IGNORE_CACHE = os.environ.get("IGNORE_CACHE", "false").lower() == "true"

# ServiceX behavior: choose query language
USE_SERVICEX_UPROOT_RAW = (
    os.environ.get("USE_SERVICEX_UPROOT_RAW", "true").lower() == "true"
)

# %% [markdown]
# ## Introduction
#
# We are going to use the ATLAS Open Data for this demonstration, in particular a $H\rigl
#
# This notebook is meant as a **technical demonstration**. In particular, the systematic
#
# This notebook implements most of the analysis pipeline shown in the following picture,
# ![ecosystem visualization](utils/ecosystem.png)

# %% [markdown]
# ### Tools and packages used in this example
#
# The notebook showcases:
# - data delivery with `ServiceX`
# - event / column selection with `func_adl`
```

```
# - data handling with `awkward-array`
# - histogram production with `coffea`
# - histogram handling with `hist`
# - visualization with `mplhep`, `hist` & `matplotlib`
# - ROOT file handling with `uproot`
# - statistical model construction with `cabinetry`
# - statistical inference with `pyhf`

# %% [markdown]
# ### High-level strategy
#
# We will define which files to process, set up a query with `func_adl` to extract data
# Those histograms will be saved with `uproot`, and then assembled into a statistical mo
# Following that, we perform statistical inference with `pyhf`.

# %% [markdown]
# ### Required setup for this notebook
#
# If you are running on the coffea-casa Open Data instance, ServiceX credentials are aut
# Otherwise you will need to set those up.
# Create a file `servicex.yaml` in your home directory, or the place this notebook is lo
#
# See this [talk by KyungEon](https://indico.cern.ch/event/1076231/contributions/4560404/

# %% [markdown]
# ## Files to process
#
# To get started, we define which files are going to be processed in this notebook.
# We also set some information for histogramming that will be used subsequently.

# %%
prefix = "http://xrootd-local.unl.edu:1094//store/user/AGC/ATLAS_HZZ/"

# labels for combinations of datasets
z_ttbar = r"Background $Z,t\bar{t}$"
zzstar = r"Background $ZZ^{\star}$"
signal = r"Signal ($m_H$ = 125 GeV)"

input_files = {
    "Data": [
        prefix + "data_A.4lep.root",
        prefix + "data_B.4lep.root",
        prefix + "data_C.4lep.root",
        prefix + "data_D.4lep.root",
    ],
    z_ttbar: [
```

```
        prefix + "mc_361106.Zee.4lep.root",
        prefix + "mc_361107.Zmumu.4lep.root",
        prefix + "mc_410000.ttbar_lep.4lep.root",
    ],
    zzstar: [prefix + "mc_363490.llll.4lep.root"],
    signal: [
        prefix + "mc_345060.ggH125_ZZ4lep.4lep.root",
        prefix + "mc_344235.VBFH125_ZZ4lep.4lep.root",
        prefix + "mc_341964.WH125_ZZ4lep.4lep.root",
        prefix + "mc_341947.ZH125_ZZ4lep.4lep.root",
    ],
}

# information for histograms
bin_edge_low = 80  # 80 GeV
bin_edge_high = 250  # 250 GeV
num_bins = 34

# %%
# Stage 1: query construction and cached dataset retrieval.
SCALE_FACTORS = [
    "scaleFactor_ELE",
    "scaleFactor_MUON",
    "scaleFactor_LepTRIGGER",
    "scaleFactor_PILEUP",
]
COLUMNS = [
    "lep_n",
    "lep_pt",
    "lep_eta",
    "lep_phi",
    "lep_E",
    "lep_charge",
    "lep_type",
    "mcWeight",
    *SCALE_FACTORS,
]
DERIVED_COLUMNS = ["scaleFactor", "scaleFactor_up", "scaleFactor_down"]
USE_SERVICEX_DOWNLOAD = True


def get_lepton_query():
    """Select four-lepton events and calculate the SF nuisance at query time."""
    return (
        servicex.query.FuncADL_Uproot()
        .FromTree("mini")
```

```
        .Where(lambda e: e.lep_n == 4)
        .Select(
            lambda e: {
                "lep_n": e.lep_n,
                "lep_pt": e.lep_pt,
                "lep_eta": e.lep_eta,
                "lep_phi": e.lep_phi,
                "lep_E": e.lep_E,
                "lep_charge": e.lep_charge,
                "lep_type": e.lep_type,
                "mcWeight": e.mcWeight,
                "scaleFactor_ELE": e.scaleFactor_ELE,
                "scaleFactor_MUON": e.scaleFactor_MUON,
                "scaleFactor_LepTRIGGER": e.scaleFactor_LepTRIGGER,
                "scaleFactor_PILEUP": e.scaleFactor_PILEUP,
                "scaleFactor": e.scaleFactor_ELE
                * e.scaleFactor_MUON
                * e.scaleFactor_LepTRIGGER
                * e.scaleFactor_PILEUP,
                "scaleFactor_up": e.scaleFactor_ELE
                * e.scaleFactor_MUON
                * e.scaleFactor_LepTRIGGER
                * e.scaleFactor_PILEUP
                * 1.1,
                "scaleFactor_down": e.scaleFactor_ELE
                * e.scaleFactor_MUON
                * e.scaleFactor_LepTRIGGER
                * e.scaleFactor_PILEUP
                * 0.9,
            }
        )
    )


def get_lepton_query_uproot_raw():
    product = "*".join(SCALE_FACTORS)
    return servicex.query.UprootRaw(
        [
            {
                "treename": "mini",
                "cut": "lep_n == 4",
                "expressions": COLUMNS + DERIVED_COLUMNS,
                "aliases": {
                    "scaleFactor": product,
                    "scaleFactor_up": f"({product})*1.1",
                    "scaleFactor_down": f"({product})*0.9",
```

```
                },
            }
        ]
    )


# MC metadata from the AGC ATLAS HZZ utils/infofile.py (xsec in pb).
# https://github.com/iris-hep/analysis-grand-challenge/blob/main/analyses/atlas-open-data
# Normalize each input independently: ServiceX filenames need not retain DSIDs.
MC_NORMALIZATION = {
    "361106": (1950.5295, 150277594200),
    "361107": (1950.6321, 147334691090),
    "410000": (452.693559, 49386600),
    "363490": (1.2578, 7538705.8077),
    "345060": (0.0060239, 27881776.6536),
    "344235": (0.0004633012, 3680490.83243),
    "341964": (0.0003769, 149400),
    "341947": (2.1424784e-06, 150000),
}
LUMINOSITY_PB = 10_000.0


def retrieve_datasets():
    query = (
        get_lepton_query_uproot_raw() if USE_SERVICEX_UPROOT_RAW else get_lepton_query()
    )
    samples, metadata = [], {}
    for category, urls in input_files.items():
        for url in urls:
            name = Path(url).name.replace(".root", "")
            meta = {"dataset_name": category}
            if category != "Data":
                dsid = re.search(r"mc_(\d+)", url).group(1)
                xsec, sumw = MC_NORMALIZATION[dsid]
                meta.update(xsec=xsec, sumw=sumw)
            metadata[name] = meta
            samples.append(
                {
                    "Name": name,
                    "Query": query,
                    "Dataset": servicex.dataset.FileList([url]),
                    "IgnoreLocalCache": IGNORE_CACHE,
                }
            )
    start = time.monotonic()
    results = servicex.deliver(
```

```
        {
            "General": {"Delivery": "LocalCache" if USE_SERVICEX_DOWNLOAD else "URLs"},
            "Sample": samples,
        }
    )
    fileset = {}
    for name, meta in metadata.items():
        paths = list(results[name])
        if not paths:
            raise RuntimeError(f"ServiceX returned no files for {name}")
        fileset[name] = {"files": paths, "metadata": meta}
        # Inspect the delivered tree and require the complete downstream schema.
        for path in paths:
            with uproot.open(path) as root_file:
                tree_name = "mini" if USE_SERVICEX_UPROOT_RAW else "servicex"
                tree = root_file[tree_name]
                missing = set(COLUMNS + DERIVED_COLUMNS) - set(tree.keys())
                if missing:
                    raise RuntimeError(f"Missing columns in {name}: {missing}")
                logger.info("Retrieved %s: %d events", name, tree.num_entries)
    logger.info("Retrieval completed in %.1f seconds", time.monotonic() - start)
    return fileset


# %%
# Stage 2: four-lepton selection, weights, mass reconstruction and histograms.
# Demonstration uncertainties, not calibrated detector/theory uncertainties.
MASS_VARIATION = 0.01
ZZ_NORMALIZATION_UNCERTAINTY = 0.10
SIGNAL_NORMALIZATION = 1.0
MASS_LABEL = r"$m_{4\ell}$ [GeV]"
MC_SAMPLES = [z_ttbar, zzstar, signal]
VARIATIONS = [
    "nominal",
    "scaleFactor_up",
    "scaleFactor_down",
    "m4l_up",
    "m4l_down",
    "ZZ_norm_up",
    "ZZ_norm_down",
]


def lepton_filter(charge, flavor):
    """Require two disjoint SFOS pairs covering all four electrons/muons.
```

```
    Balanced charge within each flavor is equivalent to the existence of a
    perfect pairing. 4e/4mu have multiple possible pairings and must be kept.
    """
    allowed = ak.all(((flavor == 11) | (flavor == 13)) & (abs(charge) == 1), axis=1)
    balanced = ak.ones_like(allowed, dtype=bool)
    for pdgid in (11, 13):
        balanced = balanced & (
            ak.sum((flavor == pdgid) & (charge == 1), axis=1)
            == ak.sum((flavor == pdgid) & (charge == -1), axis=1)
        )
    return (ak.num(charge, axis=1) == 4) & allowed & balanced


def empty_histogram():
    return (
        hist.Hist.new.Reg(
            num_bins, bin_edge_low, bin_edge_high, name="m4l", label=MASS_LABEL
        )
        .StrCat(["Data", *MC_SAMPLES], name="process")
        .StrCat(VARIATIONS, name="variation")
        .Weight()
    )


class HZZAnalysis(processor.ProcessorABC):
    def process(self, events):
        category = events.metadata["dataset_name"]
        selected = events[lepton_filter(events.lep_charge, events.lep_type)]
        leptons = ak.zip(
            {
                "pt": selected.lep_pt,
                "eta": selected.lep_eta,
                "phi": selected.lep_phi,
                "energy": selected.lep_E,
            },
            with_name="Momentum4D",
        )
        # Open Data momenta/energies are in MeV; fit and plot in GeV.
        mass = ak.to_numpy(ak.sum(leptons, axis=1).mass) / 1000.0
        if not np.all(np.isfinite(mass)):
            raise ValueError(f"Non-finite four-lepton mass in {category}")
        result = empty_histogram()
        if category == "Data":
            result.fill(
                m4l=mass,
                process=category,
```

```
                variation="nominal",
                weight=np.ones(len(mass)),
            )
        else:
            basic = (
                LUMINOSITY_PB
                * events.metadata["xsec"]
                / events.metadata["sumw"]
                * ak.to_numpy(selected.mcWeight)
            )
            if category == signal:
                basic = basic * SIGNAL_NORMALIZATION
            nominal = basic * ak.to_numpy(selected.scaleFactor)
            for variation in VARIATIONS:
                varied_mass, weight = mass, nominal
                if variation.startswith("scaleFactor_"):
                    weight = basic * ak.to_numpy(selected[variation])
                elif variation.startswith("m4l_"):
                    shift = (
                        MASS_VARIATION if variation.endswith("up") else -MASS_VARIATION
                    )
                    # Only m4l changes: the flavor/charge cut is invariant.
                    # Filling after the shift includes migration across SR edges.
                    varied_mass = mass * (1.0 + shift)
                elif variation.startswith("ZZ_norm_") and category == zzstar:
                    shift = (
                        ZZ_NORMALIZATION_UNCERTAINTY
                        if variation.endswith("up")
                        else -ZZ_NORMALIZATION_UNCERTAINTY
                    )
                    weight = nominal * (1.0 + shift)
                if not np.all(np.isfinite(weight)):
                    raise ValueError(f"Non-finite weight in {category}/{variation}")
                result.fill(
                    m4l=varied_mass,
                    process=category,
                    variation=variation,
                    weight=weight,
                )
        return {
            "histogram": result,
            "retrieved": len(events),
            "selected": len(selected),
        }

    def postprocess(self, accumulator):
```

```
        return accumulator


def process_datasets(fileset):
    """Run a coffea processor on bounded, eager BaseSchema chunks locally.

    Current coffea exposes NanoEventsFactory directly; the legacy Runner API
    is unnecessary here. Metadata stays attached per input, not per category.
    """
    analysis = HZZAnalysis()
    total = empty_histogram()
    cutflow = {}
    tree_name = "mini" if USE_SERVICEX_UPROOT_RAW else "servicex"
    for name, entry in fileset.items():
        counts = {"retrieved": 0, "selected": 0}
        for path in entry["files"]:
            with uproot.open(path) as root_file:
                nentries = root_file[tree_name].num_entries
            for start in range(0, nentries, CHUNKSIZE):
                events = NanoEventsFactory.from_root(
                    {path: tree_name},
                    schemaclass=BaseSchema,
                    mode="eager",
                    entry_start=start,
                    entry_stop=min(start + CHUNKSIZE, nentries),
                    metadata=entry["metadata"],
                ).events()
                if not ak.all(events.lep_n == 4):
                    raise ValueError("Query did not enforce lep_n == 4")
                output = analysis.process(events)
                total += output["histogram"]
                for key in counts:
                    counts[key] += output[key]
        cutflow[name] = counts
        logger.info("Processed %s: %s", name, counts)
    total = analysis.postprocess(total)
    for sample in ["Data", *MC_SAMPLES]:
        h = total[:, sample, "nominal"]
        if h.sum().value <= 0:
            raise ValueError(f"Empty signal-region yield for {sample}")
        logger.info("SR yield %s: %.4f", sample, h.sum().value)
    Path("cutflow.json").write_text(json.dumps(cutflow, indent=2) + "\n")
    return total


# %%
```

```
# Stage 3: nominal data/MC comparison, systematic overlays and ROOT export.
FIGURES = Path.cwd() / "figures"
SAMPLE_PATHS = {"Data": "data", z_ttbar: "Z_tt", zzstar: "ZZ", signal: "signal"}
COLORS = ["#85b9db", "#e9a65b", "#c84b53"]


def save_figure(figure, name):
    FIGURES.mkdir(parents=True, exist_ok=True)
    for extension in ("pdf", "png"):
        figure.savefig(FIGURES / f"{name}.{extension}", bbox_inches="tight", dpi=160)
    plt.close(figure)


def plot_histograms(histogram):
    plt.style.use(mplhep.style.ATLAS)
    plt.rcParams["mathtext.fontset"] = "dejavusans"
    fig, ax = plt.subplots(figsize=(9, 7))
    mc = [histogram[:, sample, "nominal"] for sample in MC_SAMPLES]
    mplhep.histplot(
        mc, stack=True, histtype="fill", label=MC_SAMPLES, color=COLORS, ax=ax
    )
    observed = histogram[:, "Data", "nominal"]
    mplhep.histplot(observed, histtype="errorbar", color="black", label="Data", ax=ax)
    total = sum(mc)
    sigma = np.sqrt(total.variances())
    edges = total.axes[0].edges
    ax.stairs(
        total.values() + sigma,
        edges,
        baseline=total.values() - sigma,
        fill=True,
        facecolor="none",
        hatch="////",
        edgecolor="gray",
        label="MC stat.",
    )
    ax.set(
        xlabel=MASS_LABEL, ylabel="Events / 5 GeV", title="Four-lepton signal region"
    )
    ax.legend(fontsize=12)
    save_figure(fig, "m4l_signal_region")

    # Each nuisance gets a comparison figure; show every affected MC sample.
    for nuisance in ("scaleFactor", "m4l", "ZZ_norm"):
        samples = [zzstar] if nuisance == "ZZ_norm" else MC_SAMPLES
        fig, axes = plt.subplots(
```

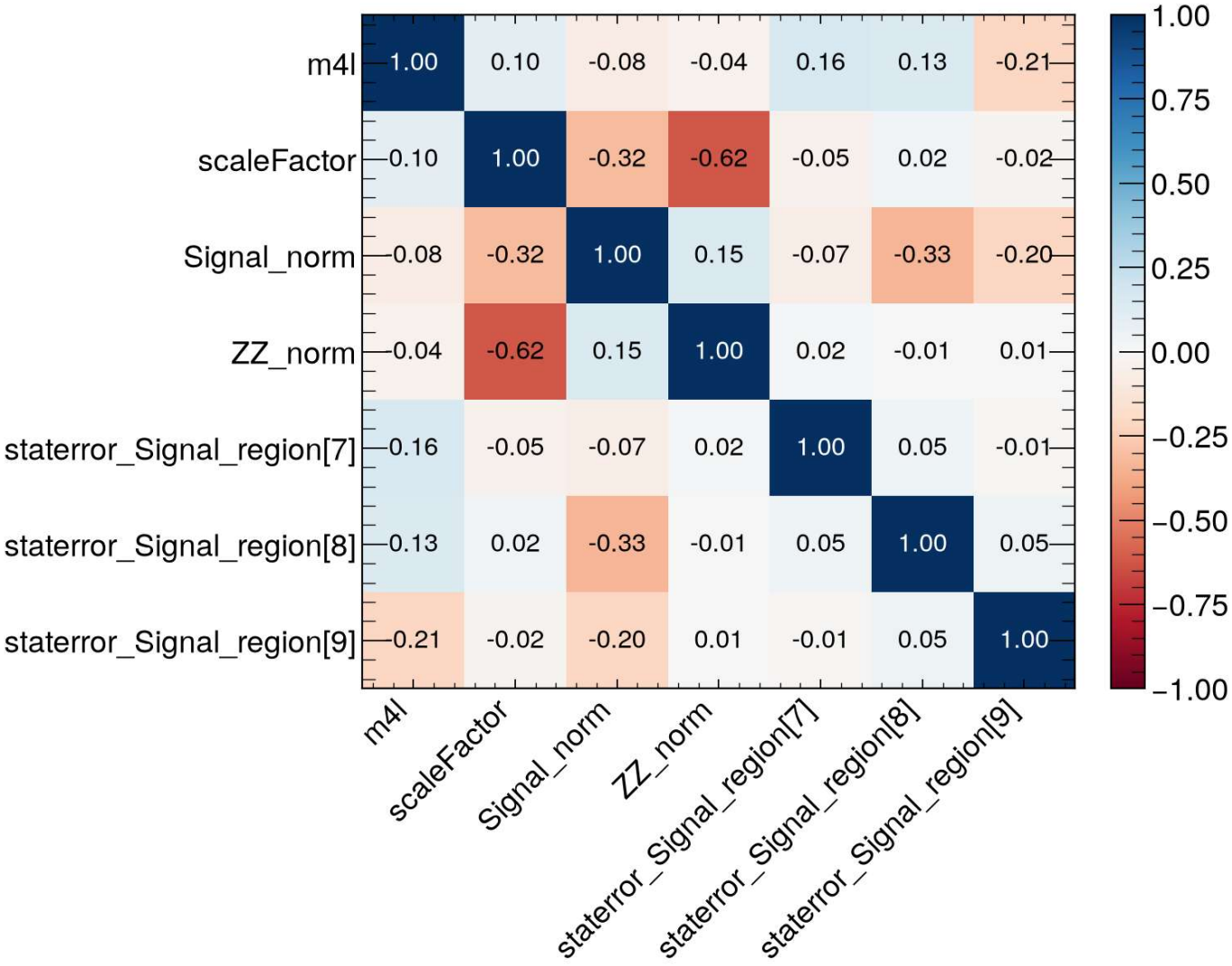
```
            1, len(samples), figsize=(7 * len(samples), 5), squeeze=False
        )
        for ax, sample in zip(axes[0], samples):
            for variation, label, color in [
                ("nominal", "Nominal", "black"),
                (f"{nuisance}_up", "Up", "#c84b53"),
                (f"{nuisance}_down", "Down", "#347fb4"),
            ]:
                mplhep.histplot(
                    histogram[:, sample, variation],
                    histtype="step",
                    yerr=False,
                    label=label,
                    color=color,
                    ax=ax,
                )
            ax.set(xlabel=MASS_LABEL, ylabel="Events / 5 GeV", title=sample)
            ax.legend(fontsize=12)
        fig.suptitle(f"{nuisance} variation")
        fig.tight_layout()
        save_figure(fig, f"systematic_{nuisance}")

    # ROOT names match the cabinetry VariationPath suffixes exactly.
    with uproot.recreate("histograms.root") as output:
        for sample, path in SAMPLE_PATHS.items():
            variations = ["nominal"] if sample == "Data" else VARIATIONS
            for variation in variations:
                suffix = "" if variation == "nominal" else f"_{variation}"
                output[path + suffix] = histogram[:, sample, variation]
    logger.info("Saved nominal/systematic figures and histograms.root")


# %%
# Stage 4: cabinetry configuration, templates, workspace and signal-strength fit.
POI = "Signal_norm"
CONFIG_SAMPLES = {
    "Data": "Data",
    signal: "Signal",
    zzstar: "Background ZZ*",
    z_ttbar: "Background Z, ttbar",
}


def statistical_analysis(histogram):
    config_path = Path("config.yml")
    if not config_path.exists():
```

```
        config = {
            "General": {
                "Measurement": "CabinetryHZZAnalysis",
                "POI": POI,
                "HistogramFolder": "histograms",
                "InputPath": "histograms.root:{SamplePath}{VariationPath}",
                "VariationPath": "",
            },
            "Regions": [
                {
                    "Name": "Signal_region",
                    "Variable": MASS_LABEL,
                    "Binning": histogram.axes[0].edges.tolist(),
                }
            ],
            "Samples": [
                dict(
                    Name=CONFIG_SAMPLES[sample],
                    SamplePath=path,
                    **({"Data": True} if sample == "Data" else {}),
                )
                for sample, path in SAMPLE_PATHS.items()
            ],
            "Systematics": [
                {
                    "Name": nuisance,
                    "Type": "NormPlusShape",
                    "Samples": [CONFIG_SAMPLES[s] for s in MC_SAMPLES],
                    "Up": {"VariationPath": f"_{nuisance}_up"},
                    "Down": {"VariationPath": f"_{nuisance}_down"},
                }
                for nuisance in ("scaleFactor", "m4l")
            ]
            + [
                {
                    "Name": "ZZ_norm",
                    "Type": "Normalization",
                    "Samples": "Background ZZ*",
                    "Up": {"Normalization": ZZ_NORMALIZATION_UNCERTAINTY},
                    "Down": {"Normalization": -ZZ_NORMALIZATION_UNCERTAINTY},
                }
            ],
            "NormFactors": [
                {
                    "Name": POI,
                    "Samples": "Signal",
```

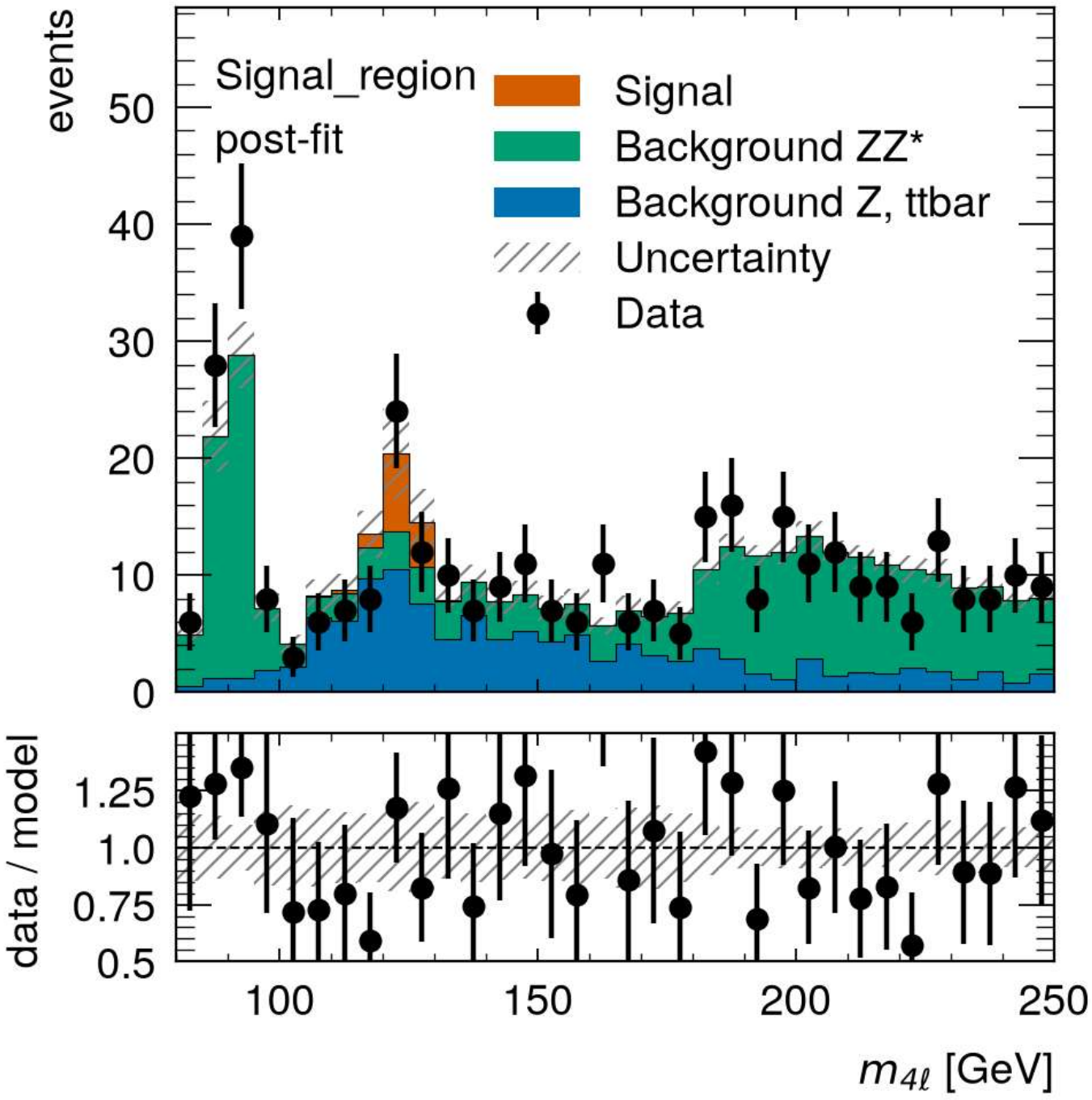
```
                    "Nominal": 1.0,
                    "Bounds": [0.0, 10.0],
                }
            ],
        }
        config_path.write_text(yaml.safe_dump(config, sort_keys=False))
    config = cabinetry.configuration.load(config_path)
    logger.info("Fit samples: %s", config["Samples"])
    logger.info("Fit systematics: %s", config["Systematics"])
    cabinetry.templates.collect(config)
    cabinetry.templates.postprocess(config)
    workspace = cabinetry.workspace.build(config)
    cabinetry.workspace.save(workspace, "workspace.json")

    # Fit observed counts, with finite-template statistical uncertainties included.
    model, data = cabinetry.model_utils.model_and_data(workspace)
    fit_results = cabinetry.fit.fit(model, data)
    poi_index = fit_results.labels.index(config["General"]["POI"])
    result = {
        "poi_val": float(fit_results.bestfit[poi_index]),
        "poi_err": float(fit_results.uncertainty[poi_index]),
    }
    if not np.all(np.isfinite(list(result.values()))) or result["poi_err"] <= 0:
        raise RuntimeError(f"Invalid fit result: {result}")

    # Pulls.
    fig = cabinetry.visualize.pulls(
        fit_results,
        exclude=config["General"]["POI"],
        close_figure=False,
        save_figure=False,
    )
    save_figure(fig, "pull_plot")

    # Parameter correlations.
    fig = cabinetry.visualize.correlation_matrix(
        fit_results, pruning_threshold=0.15, close_figure=False, save_figure=False
    )
    save_figure(fig, "matrix")

    # Post-fit data/model comparison, with the original mass binning and units.
    prediction = cabinetry.model_utils.prediction(model, fit_results=fit_results)
    figures = cabinetry.visualize.data_mc(
        prediction, data, config=config, close_figure=False, save_figure=False
    )
    for item in figures:
```

```
        fig = item["figure"]
        fig.axes[-1].set_xlabel(MASS_LABEL)
        save_figure(fig, "model-comp")

    # Write the requested result only after all fit products have been saved.
    Path("results.json").write_text(
        json.dumps(result, indent=2, allow_nan=False) + "\n"
    )
    logger.info(
        "Signal normalization: %.6f +/- %.6f", result["poi_val"], result["poi_err"]
    )
    return result


# %%
# Command-line entry point; --stage runs all stages up to the requested one.
def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument(
        "--stage", choices=["retrieve", "process", "plot", "fit"], default="fit"
    )
    parser.add_argument(
        "--self-test",
        action="store_true",
        help="Run local physics/query tests without retrieval",
    )
    args = parser.parse_args()
    if args.self_test:
        import unittest

        suite = unittest.defaultTestLoader.discover(
            str(Path(__file__).parent), pattern="test_analysis.py"
        )
        result = unittest.TextTestRunner(verbosity=2).run(suite)
        raise SystemExit(0 if result.wasSuccessful() else 1)
    fileset = retrieve_datasets()
    if args.stage == "retrieve":
        return
    histogram = process_datasets(fileset)
    if args.stage == "process":
        return
    plot_histograms(histogram)
    if args.stage == "plot":
        return
    statistical_analysis(histogram)
```

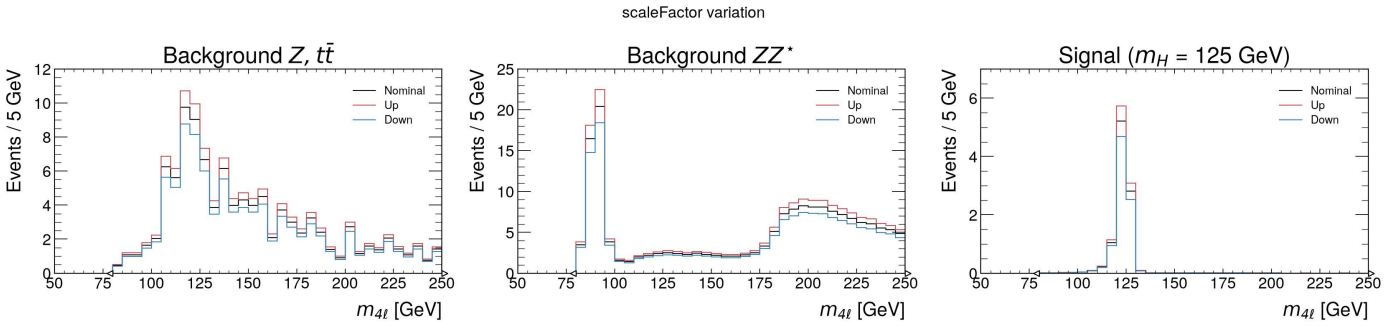
```
if __name__ == "__main__":
    main()
```



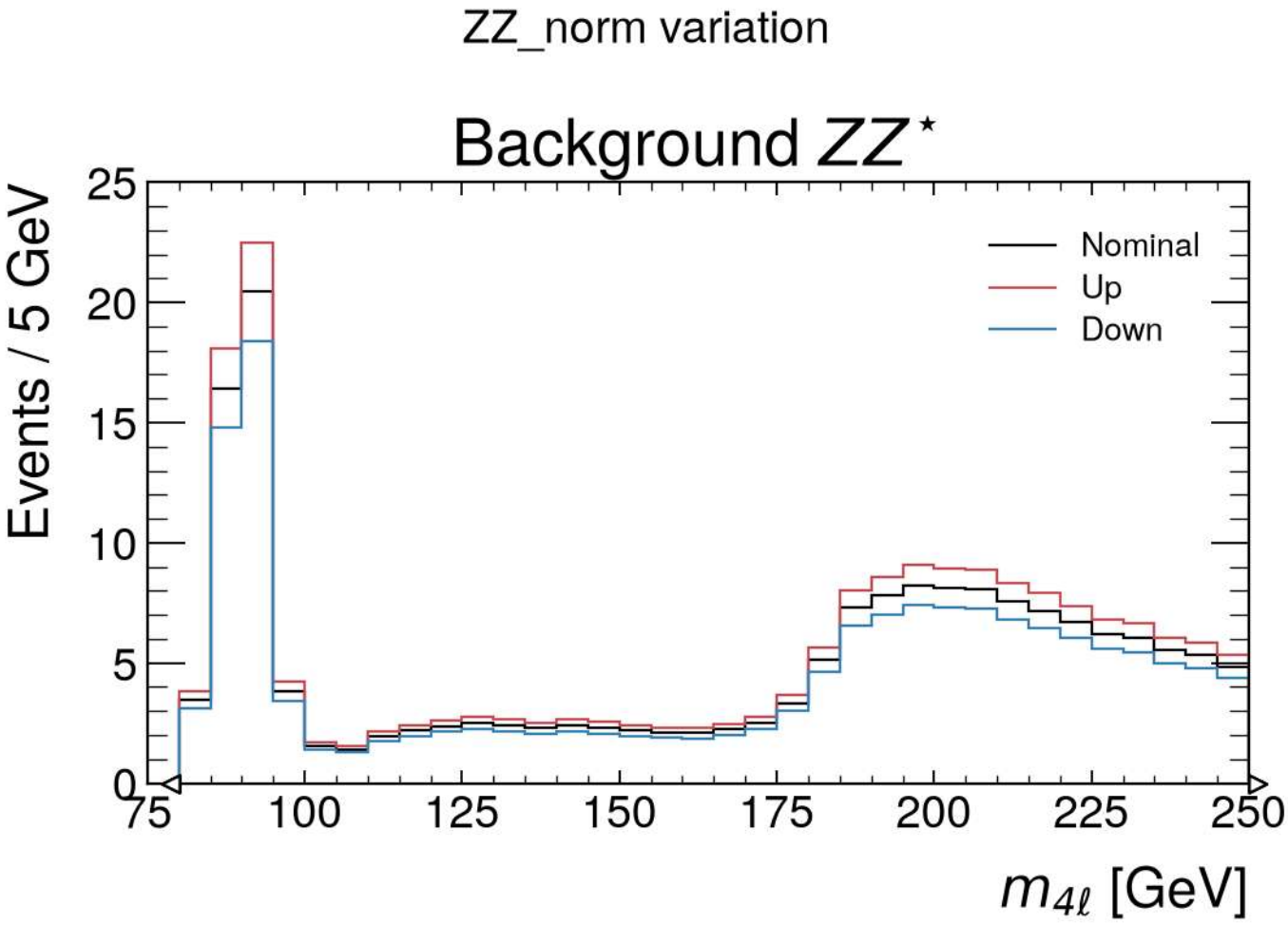
- Source: `figures/matrix.png`
- Size: 209,821 bytes



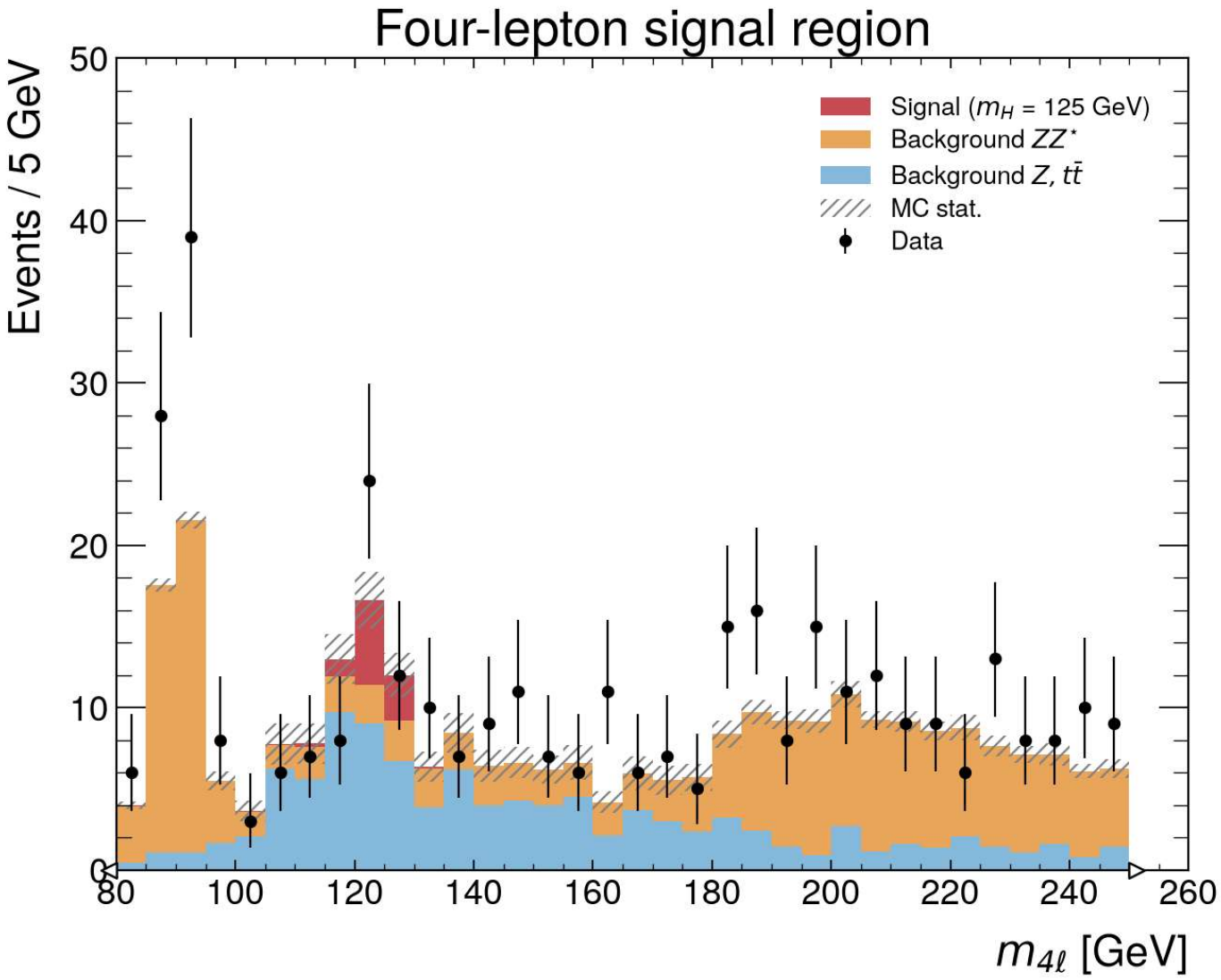
- Source: `figures/model-comp.png`
- Size: 121,685 bytes



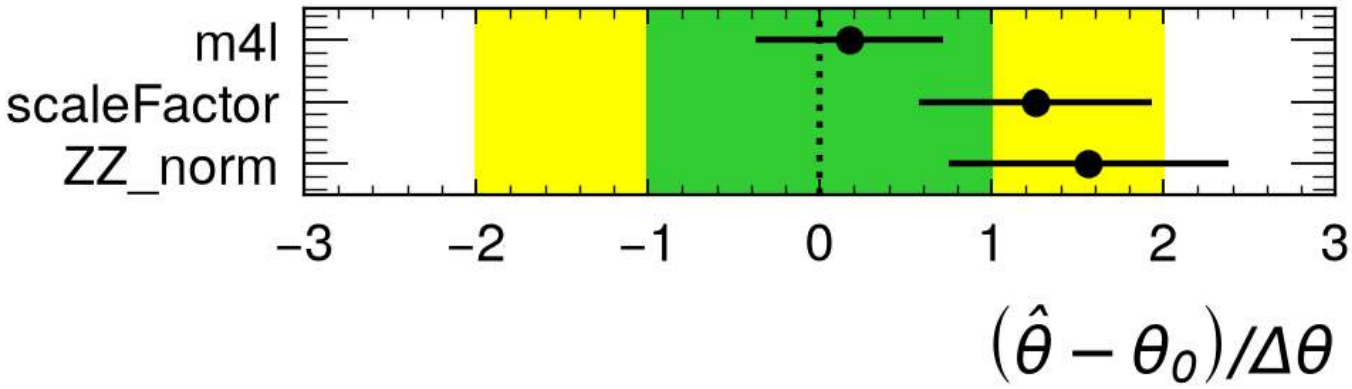
- Source: `figures/systematic_scaleFactor.png`
- Size: 173,787 bytes



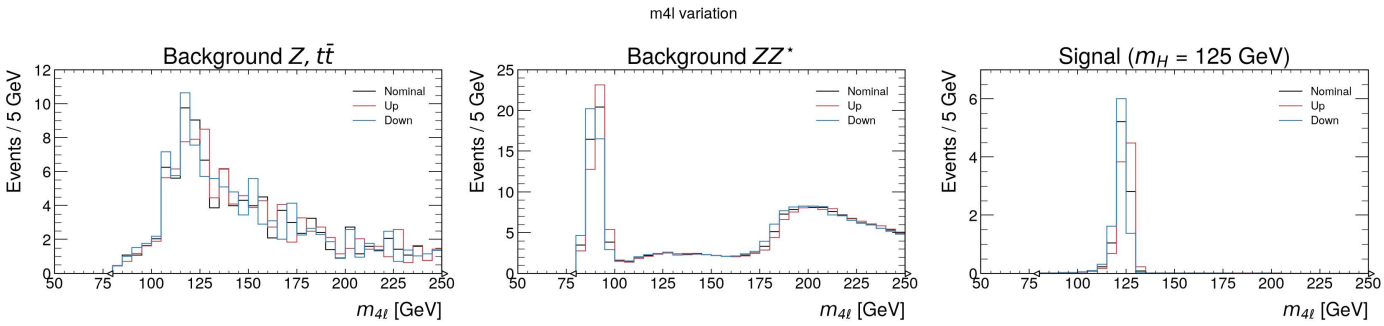
- Source: `figures/systematic_ZZ_norm.png`
- Size: 66,199 bytes



- Source: `figures/m4l_signal_region.png`
- Size: 99,077 bytes



- Source: `figures/pull_plot.png`
- Size: 26,909 bytes



- Source: `figures/systematic_m4l.png`
- Size: 170,977 bytes

▶ Complete Codex stderr

▶ Model activity

Schema version: `2`