Speedtronic Documentation

GPU-agnostic PyTorch training, documented from source

Version `2.0.0`

Generated 2026-09-25

`github.com/bench-labs-org/Speedtronic`

`speedtronic-docs.pages.dev`

Contents

Speedtronic

GPU-agnostic PyTorch training, built for speed.

Speedtronic 2.0.0 is an alpha, architecture-neutral Python training framework. A run is assembled from a validated configuration, a registered `torch.nn.Module`, a data source, AdamW or hybrid Muon, an optional scheduler, precision handling, checkpoints, metrics, and—optionally—a Hub-backed DumbDiLoCo coordinator.

> **DOCUMENTATION SCOPE**
>
> This site documents every local implementation module, every top-level class and function, every method, the v1 and v2 repository tests, both shipped configurations, both examples, and the packaging surface. The generated source inventory is rebuilt from the Python AST during the documentation build. Documentation coverage is not the same as runtime test coverage; see the test map.

Capability map

Area	What Speedtronic 2.0.0 provides
Devices	CPU, CUDA, and Apple MPS selection
Model contract	Any module following Speedtronic's loss/batch protocol
Reference model	Decoder-only transformer with RoPE, GQA, SwiGLU, RMSNorm, and tied embeddings
Optimization	AdamW, hybrid Muon/Muon+, cautious updates, gradient accumulation, clipping, warmup, cosine or constant schedule
Precision	FP32, FP16, BF16, CUDA `GradScaler`, fused-AdamW probing
Performance	Optional `torch.compile`, opt-in CUDA stream backprop, gradient checkpointing, worker prefetch, pinned memory
Data	Deterministic synthetic data, streamed UTF-8 text, serialized datasets, programmatic datasets
Persistence	Atomic local checkpoints with model/optimizer/scheduler/RNG/counters
Distributed	Asynchronous DumbDiLoCo over a Hugging Face model repository
Observability	Text/JSONL metrics plus callable, W&B, and TensorBoard hook adapters

One-minute path

1. Install and run the CPU smoke test.
2. Read core concepts.
3. Choose a tutorial: custom model, custom data, performance, checkpointing, or observability.
4. Use the configuration reference and complete source inventory while integrating.

System shape

Documentation map

Learn the system

- Architecture explains composition, training order, and state ownership.
- Runtime and Trainer documents the public training API and its contracts.
- Configuration lists every field, default, normalization rule, and precedence edge case.

Explore v2

- v2 overview
- Muon, Muon+, and cautious updates
- Out-of-order backprop scheduling
- Shape validation
- Migration and deferred decisions

Build an integration

- Custom models
- Custom data
- Observability hooks
- DumbDiLoCo

Operate the framework

- Testing
- Packaging
- Troubleshooting
- Security and limitations

Project identity

- Package/distribution name: `speedtronic`
- Version: `2.0.0`
- Python: `>=3.10`
- PyTorch: `>=2.1`
- License: Apache-2.0

- Default training device: `auto` → CUDA, then MPS, then CPU
- Default checkpoint directory: `<run.output_dir>/checkpoints`
- Distributed transport: Hugging Face model repository files

Local quickstart

This path runs entirely from a source checkout. It uses synthetic data, does not download a model or dataset, and does not require Hugging Face credentials.

Prerequisites

- Python 3.10 or newer
- PyTorch 2.1 or newer
- A writable checkout
- For CPU-only installation, install an appropriate PyTorch CPU wheel first when needed

Install from source

```
cd /path/to/speedtronic
python3 -m venv .venv
. .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install -e '.[dev]'
```

The package installs the `speedtronic` console command. The equivalent module entry point is:

```
python -m speedtronic --help
```

Validate the configuration

```
speedtronic validate --config configs/smoke.yaml
```

Validation parses and normalizes configuration. It does **not** instantiate the model, open the selected data, resolve hardware, or run a forward pass.

The effective smoke configuration has these important properties:

Property	Value
Target	4 global optimizer steps
Model	2-layer reference transformer, 66,880 parameters
Data	Deterministic synthetic token blocks
Microbatch	1 sample
Target batch	2 samples
Accumulation	2 microbatches per optimizer update
Precision	FP32
Device	`auto` in YAML; pass `--device cpu` to force it
Checkpoint interval	Every 2 steps
Retention	Last 2 checkpoints

Run four steps

```
speedtronic train --config configs/smoke.yaml --device cpu
```

A successful run ends with output similar to:

```
completed steps=4 samples=8 tokens=256 loss=<value>
```

Loss is stochastic for synthetic input and initialization state, so the exact value is not contractual.

Artifacts are written beneath:

```
runs/smoke/checkpoints/
├── latest.json
├── step_000000000002.pt
└── step_000000000004.pt
```

The pointer is:

```
{"step": 4, "file": "step_000000000004.pt"}
```

Resume semantics

`max_steps` is an absolute global target, not a count of additional work:

```
speedtronic train --config configs/smoke.yaml --device cpu --resume
```

If the newest checkpoint is at step 4, this invocation performs no optimizer updates. To demonstrate resume, use a temporary output directory, run two steps, then resume to four:

```
speedtronic train \
  --config configs/smoke.yaml \
  --device cpu \
  --output-dir runs/resume-demo \
  --max-steps 2

speedtronic train \
  --config configs/smoke.yaml \
  --device cpu \
  --output-dir runs/resume-demo \
  --max-steps 4 \
  --resume
```

The second command reports cumulative `steps=4`, even though it performed two updates.

Python equivalent

```python
from speedtronic.runtime import train_from_config

result = train_from_config(
    {
        "run": {
            "name": "demo",
            "max_steps": 4,
            "device": "cpu",
            "output_dir": "runs/python-demo",
        },
        "model": {
            "name": "reference_transformer",
            "vocab_size": 128,
            "max_seq_len": 32,
            "n_layer": 2,
            "n_head": 4,
            "n_kv_head": 2,
            "d_model": 64,
            "d_ff": 128,
        },
        "data": {
            "micro_batch_size": 1,
            "target_batch_size": 2,
            "num_tokens": 32,
        },
        "precision": {"mode": "fp32"},
        "scheduler": {"warmup_steps": 1, "max_steps": 4},
    }
)

print(result.steps, result.samples, result.tokens, result.final_loss)
```

CONFIGURE THE SCHEDULER HORIZON EXPLICITLY

`SchedulerConfig.max_steps` defaults to 1000. When a shorter `run.max_steps` is set and the scheduler is left at that default, Speedtronic 2.0 automatically adopts the run target so the schedule ends with training. Set `scheduler.max_steps` explicitly when you want a horizon that differs from the run target.

What the four steps execute

For each global step:

1. Fetch two synthetic microbatches.
2. Run each through the reference model in FP32.
3. Compute a causal-language-model loss.
4. Backpropagate each loss divided by two.
5. Clip only if `optimizer.grad_clip` is configured.
6. Perform one AdamW update.

7. Advance the cosine scheduler.
8. Emit `train_step` metrics.
9. Save a checkpoint at steps 2 and 4.

Continue with core concepts or the architecture reference.

Core concepts

Configuration is the composition contract

`SpeedtronicConfig` is a tree of mutable dataclasses:

```
SpeedtronicConfig
├── run
├── model
├── data
├── optimizer
├── scheduler
├── precision
├── checkpoint
├── logging
├── distributed
├── gradient_checkpointing
└── compile
```

The runtime composition root, `build_runtime()`, converts that declarative description into concrete objects. Configuration validation is deliberately structural and lightweight; it does not guarantee that a data path exists or that a custom model can train.

Microbatch versus optimizer step

A loader emits `data.micro_batch_size` examples. The trainer consumes:

```
accumulation_steps = target_batch_size / micro_batch_size
```

microbatches before one optimizer update. Every loss is divided by that count before backward propagation. The scheduler, global step counter, coordinator callback, and checkpoint cadence advance once per complete optimizer update.

If the final emitted microbatch is smaller than `micro_batch_size` and `drop_last=False`, the update can contain fewer than the nominal target batch.

Absolute global steps

`run.max_steps`, `data.max_steps`, the `Trainer` constructor, and `Trainer.fit()` describe an absolute target:

```
target_step = max(self.step, requested_target)
```

A trainer resumed at step 900 with a target of 1000 performs 100 steps. A trainer already at step 1000 performs none and returns `final_loss=None`.

Model input contract

The built-in trainer recognizes two batch forms:

Batch form	Forward call
`dict`	`model(**batch)`
`tuple` or `list` with at least two elements	`model(batch[0], batch[1])`

The built-in causal loader emits:

```
{
    "input_ids": LongTensor[batch, sequence],
    "labels": LongTensor[batch, sequence],
    "attention_mask": BoolTensor[batch, sequence],
}
```

A custom dataset that returns arbitrary dictionaries does not automatically receive a general-purpose collator; the bundled collator specifically understands causal dictionaries.

Model output contract

A model can return:

- a scalar loss tensor;
- `{"loss": loss, ...}`;
- `{"logits": logits, ...}` plus compatible labels;
- a tuple/list whose first item is a loss;
- a tuple/list whose first item is 3-D causal logits.

A bare tensor is always interpreted as a loss, not logits. Extra metrics returned in an output mapping are not currently forwarded to `MetricLogger`.

Precision resolution

`precision.mode: auto` means:

- CUDA with BF16 support → BF16;

- CUDA without BF16 support → FP16 plus `GradScaler`;
- CPU or MPS → FP32.

Explicit FP16 on CPU and mixed precision on MPS fall back to FP32. Unsupported BF16 on CUDA also falls back. Explicit CPU BF16 is not capability-checked by Speedtronic.

Checkpoint versus distributed global state

A local trainer checkpoint contains model, optimizer, scheduler, scaler, counters, RNG, redacted config, and coordinator state. It is saved beneath the local filesystem.

DumbDiLoCo additionally maintains:

- a Hub-global model and outer step;
- a master's local processed-delta ledger and Nesterov momentum;
- per-node baselines and cached global files.

These are separate state domains. A worker restart begins from the latest global version it can read, not from an exact continuation of its DataLoader or partial inner loop.

DumbDiLoCo vocabulary

Term	Meaning in Speedtronic
Local step	One completed local AdamW optimizer update
Inner step	A local step; the name emphasizes that it belongs to the local objective
Inner boundary	`local_step % inner_steps == 0`
Baseline	Model state captured at the last successful upload or global installation
Pseudo-gradient	`baseline - current`; not an autograd gradient
Outer round	One successful master aggregation/publication
Outer step	Monotonic integer published in `global/step_count.json`
Node delta	Floating-state safetensors plus path/header metadata
Global model	Complete CPU-cloned `state_dict` published by the master

State ownership

State	Primary owner	Local checkpoint	Hub
Parameters and buffers	`torch.nn.Module`	Yes	Global model
AdamW moments	`torch.optim.AdamW`	Yes	No
Scheduler	`LambdaLR`	Yes	No
CUDA scaler	`GradScaler`	Yes	No
Step/sample/token counters	`Trainer`	Yes	No
Global RNG	Python/NumPy/PyTorch	Yes	No
DataLoader position	PyTorch loader/workers	No	No
Global outer model	`MasterOuterLoop`	Master state	Yes
Processed deltas	`MasterOuterLoop`	Master state	No
Nesterov momentum	`MasterOuterLoop`	Master state	No
Worker baseline	Coordinator	Yes	No

Extension seams

- **Model factory:** `ModelRegistry` / `register_model`
- **Gradient checkpointing:** `model.set_gradient_checkpointing(enabled)`
- **Dataset and tokenizer:** `build_dataloader(dataset=..., tokenizer=...)`
- **Coordinator:** implement the `SyncCoordinator` protocol
- **Metrics:** callables or objects with `on_event(event, payload)`

See architecture, extensions, and DumbDiLoCo.

Bring your own model

The training loop is architecture-neutral. A custom model must follow the batch and loss protocols and be constructible through a registry factory.

Step 1: Follow the batch contract

Dictionary batches are passed as keyword arguments. The bundled loader emits:

```
input_ids
labels
attention_mask
```

A tuple/list batch is passed as:

```
model(batch[0], batch[1])
```

Step 2: Return a usable loss

The model can return:

```
loss_tensor
```

```
{"loss": loss_tensor, "auxiliary": ...}
```

```
{"logits": logits, ...}
```

or a tuple/list whose first element is a loss or compatible 3-D logits.

A bare tensor is always interpreted as a loss, not logits.

Step 3: Register a factory

```
import torch
from torch import nn
from torch.nn import functional as F

from speedtronic import register_model


class SmallCausalLM(nn.Module):
    def __init__(self, vocab_size: int, d_model: int):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.output = nn.Linear(d_model, vocab_size)

    def forward(
        self,
        input_ids,
        labels=None,
        attention_mask=None,
    ):
        logits = self.output(self.embedding(input_ids))
        if labels is None:
            return {"logits": logits}

        if labels.shape[1] == input_ids.shape[1]:
            labels = labels[:, 1:]

        loss = F.cross_entropy(
            logits[:, :-1].reshape(-1, logits.shape[-1]),
            labels.reshape(-1),
            ignore_index=-100,
        )
        return {"logits": logits, "loss": loss}


@register_model("small_causal_lm")
def make_small_causal_lm(**kwargs):
    return SmallCausalLM(
        vocab_size=kwargs["vocab_size"],
        d_model=kwargs["d_model"],
    )
```

Factories receive the standard model fields even when they are unrelated to the custom architecture. Accepting `**kwargs` avoids accidental construction errors.

Step 4: Train programmatically

```
from speedtronic.runtime import train_from_config

result = train_from_config(
    {
        "run": {
            "max_steps": 10,
            "device": "cpu",
            "output_dir": "runs/small-lm",
        },
        "model": {
            "name": "small_causal_lm",
            "vocab_size": 128,
            "max_seq_len": 32,
            "n_layer": 2,
            "n_head": 4,
            "n_kv_head": 2,
            "d_model": 64,
        },
        "data": {
            "block_size": 32,
            "micro_batch_size": 2,
            "target_batch_size": 4,
        },
        "scheduler": {"warmup_steps": 1, "max_steps": 10},
        "precision": {"mode": "fp32"},
    }
)
```

CLI limitation

Registry entries are in-memory. A separate `speedtronic train --config ...` process does not import arbitrary application registration code. YAML can select a registered name, but it cannot discover a Python factory by itself.

For a CLI-visible custom model, add an application-owned import/plugin mechanism and construct the runtime after registration.

Non-language-model model

A model can ignore causal keys by accepting them explicitly and returning any differentiable scalar loss:

```
class TinyRegressor(nn.Module):
    def __init__(self):
        super().__init__()
        self.value = nn.Parameter(torch.zeros(()))

    def forward(self, input_ids, labels=None, attention_mask=None):
        if labels is None:
            raise ValueError("labels are required")
        return (self.value - labels.float().mean()) ** 2
```

The built-in collator still produces language-model-shaped batches, so use a custom DataLoader for other batch structures.

Direct Trainer injection

For complete control:

```
from speedtronic.trainer import Trainer

trainer = Trainer(
    model,
    optimizer,
    dataloader,
    device="cpu",
    max_steps=100,
)
result = trainer.fit()
```

Gradient checkpointing hook

```
class MyModule(nn.Module):
    def set_gradient_checkpointing(self, enabled: bool = True):
        self.use_checkpointing = bool(enabled)
```

The trainer warns rather than fails if the hook is absent or raises.

Model-returned metrics

The trainer currently discards additional output-dictionary metrics. If metrics are required, send them through a hook, a global collector, or a custom Trainer implementation.

Checklist

- [ ] Model accepts the actual batch keys.
- [ ] Output begins with a differentiable loss under the trainer's interpretation.
- [ ] Factory accepts standard registry keywords.
- [ ] Registration occurs in the same process as runtime construction.
- [ ] Model context and data block sizes are compatible.
- [ ] Custom batch shapes have a matching collator.
- [ ] Gradient checkpointing exposes the optional hook.

Related: Extensions, Data, and Runtime and Trainer.

Bring your own data

Speedtronic supports several data paths, but the built-in collator is intentionally small. Choose the path that matches the model's batch contract.

Synthetic data

Synthetic data is the default and requires no download:

```yaml
data:
  synthetic: true
  num_tokens: 128
  block_size: 32
  micro_batch_size: 2
  target_batch_size: 8
```

Each sample contains `block_size` input IDs and `block_size` next-token labels. The field `num_tokens` is passed as the sample count in the current implementation.

Text file

```yaml
data:
  text_path: /absolute/path/to/train.txt
  block_size: 128
  micro_batch_size: 1
  target_batch_size: 8
  num_workers: 0
```

The bundled reader:

- reads UTF-8 incrementally;
- uses a deterministic byte tokenizer by default;
- emits full blocks and one optional short final block;
- pads shorter final samples to the longest sample in a batch.

Use zero workers unless the dataset itself shards by worker. The bundled stream does not.

Custom tokenizer

Programmatic only:

```
class Tokenizer:
    def encode(self, text: str):
        return my_integer_ids(text)


loader = build_dataloader(
    config.data,
    tokenizer=Tokenizer(),
    vocab_size=config.model.vocab_size,
)
```

A tokenizer should support incremental or boundary-safe behavior. The bundled stream carries token IDs across reads rather than raw text, which can split context-sensitive tokenizers.

Injected Dataset

```
from torch.utils.data import Dataset

from speedtronic.data import build_dataloader


class MyDataset(Dataset):
    def __len__(self):
        return 1000

    def __getitem__(self, index):
        return {
            "input_ids": input_ids_tensor,
            "labels": labels_tensor,
        }


loader = build_dataloader(
    config.data,
    dataset=MyDataset(),
    vocab_size=config.model.vocab_size,
)
```

An explicit Dataset takes precedence over `text_path` and synthetic fallback.

Built-in collator support

`collate_batch()` currently handles:

Dataset item	Collation
Causal dictionary	Pads `input_ids`, `labels`, and `attention_mask`
Equal-width tuple/list	Stacks equal-shaped tensor fields
Tensor	Stacks into a batch
Other	Returns the list unchanged

For arbitrary dictionaries, build the DataLoader with a custom `collate_fn` and inject the resulting loader into `Trainer`.

Custom task example

```
from torch.utils.data import DataLoader, Dataset


class PixelDataset(Dataset):
    def __getitem__(self, index):
        return pixels[index], targets[index]


def pixel_collate(items):
    inputs = torch.stack([item[0] for item in items])
    targets = torch.stack([item[1] for item in items])
    return inputs, targets


loader = DataLoader(
    PixelDataset(),
    batch_size=8,
    collate_fn=pixel_collate,
)
```

The model and loss can then be task-specific.

Serialized Dataset

YAML can name an existing file:

```
data:
  synthetic: false
  dataset: /trusted/path/dataset.pt
```

The file must deserialize to a `Dataset` or `IterableDataset`. It is loaded with `weights_only=False`, so only load trusted files.

Source-selection order

```
explicit dataset argument
→ text_path
→ Dataset object in config.dataset
→ serialized dataset path
→ synthetic fallback
→ error
```

Workers, prefetch, and pinning

```yaml
data:
  num_workers: 2
  prefetch_factor: 4
  pin_memory: true
```

When workers are enabled, the loader uses persistent workers. `shuffle` is disabled automatically for iterable datasets; implement stream-local shuffling if needed.

Infinite batching

A finite loader is cycled forever. An entirely empty pass raises:

```
RuntimeError: data loader produced no batches
```

Runtime vocabulary behavior

`build_runtime()` passes `model.vocab_size` explicitly to `build_dataloader()`. Therefore, a non-default `data.vocab_size` is ignored through the standard config path. Keep model and data vocabularies aligned or use direct loader construction.

Checkpoint position

DataLoader position is not checkpointed. Resume restarts iteration and may repeat or skip examples relative to an uninterrupted process, particularly with `shuffle=False`.

Checklist

- [ ] Dataset returns batches the model accepts.
- [ ] Labels are differentiable-loss compatible.
- [ ] Custom collator handles variable shapes.
- [ ] Iterable datasets shard workers if enabled.

- [ ] Tokenizer is safe at stream boundaries.
- [ ] Dataset files and checkpoints come from trusted sources.
- [ ] Model/data vocabulary is consistent.

Related: Data API, Custom Model, and Checkpoint Resume.

Local performance

This page describes mechanisms implemented in Speedtronic 2.0.0 and their trade-offs. Repository tests are CPU-focused and do not validate every hardware path.

Gradient accumulation

```yaml
data:
  micro_batch_size: 1
  target_batch_size: 8
```

This produces eight microbatches per AdamW update. Each loss is divided by eight before backward propagation.

Accumulation:

- keeps per-microbatch parameter memory bounded;
- increases wall time for the same examples;
- allows the nominal target batch to exceed loader batch size;
- still counts one global optimizer step and one scheduler step.

The final microbatch may be smaller when `drop_last=False`.

Precision

Mode	Typical use
`fp32`	Maximum portability and numerical stability
`bf16`	CUDA devices with native BF16 support
`fp16`	CUDA fallback with dynamic loss scaling
`auto`	BF16, FP16, or FP32 based on device

See Precision for the exact matrix and caveats.

DataLoader throughput

```yaml
data:
  num_workers: 2
  prefetch_factor: 2
  pin_memory: true
```

- Workers run in separate processes and can overlap data preparation.
- Persistent workers remain alive between loader epochs.
- Pinning accelerates CPU-to-CUDA transfers.
- Iterable datasets cannot be shuffled by a sampler.
- The bundled text stream duplicates full-file work across workers.

Benchmark worker changes rather than assuming monotonic speedup.

Fused AdamW

```yaml
optimizer:
  fused: null
```

Speedtronic probes fused AdamW on CUDA and falls back if construction fails. The model is built on CPU before being moved, so the current probe and actual construction can disagree about device placement.

`torch.compile`

```yaml
compile: true
```

or:

```yaml
compile:
  enabled: true
```

Compilation is best-effort. Construction failure continues eagerly. A runtime exception from a compiled forward disables compilation and retries the same batch eagerly.

The broad runtime catch can mask a model bug if eager execution succeeds, and rerunning can duplicate RNG or side effects. Enable it after establishing a correct eager baseline.

Gradient checkpointing

```
gradient_checkpointing: true
```

The model must expose:

```
set_gradient_checkpointing(enabled: bool)
```

The reference transformer propagates the flag to every block.

Trade-off:

- lower activation memory;
- additional forward recomputation;
- potentially lower throughput;
- smaller feasible model/batch sizes.

Reference attention

The reference model uses `torch.nn.functional.scaled_dot_product_attention`, allowing PyTorch to choose flash, memory-efficient, or math kernels without a separate attention package.

Supplying `attention_mask` creates a dense `(B, 1, T, T)` floating mask. This can reduce kernel efficiency. With no mask, the model uses `is_causal=True`.

Scheduler and measured LR

The optimizer updates, then the scheduler advances, then metrics read the current LR. The logged LR is therefore the next step's LR, not the rate that produced the just-completed update.

Throughput synchronization

Every microbatch executes:

```
float(loss.detach().float().cpu())
```

On CUDA, this synchronizes the device on each microbatch. Token counting with an attention mask also performs `.sum().item()`. These operations can reduce throughput in a speed-focused engine.

Metric memory queries can synchronize as well.

Fused outer operations

DumbDiLoCo outer aggregation occurs in FP32 on CPU. The Hub upload/download path is I/O-heavy and can dominate outer rounds for small deltas or large models.

v2 additions

- Hybrid Muon changes the optimizer algorithm, not just kernel scheduling.
- OOO backprop is opt-in and CUDA-specific in its active path.
- Shape validation warns about unaligned effective dimensions.

Muon, Muon+, and cautious updates can compose with local performance settings, but benchmark convergence as well as throughput.

Recommended optimization order

1. Establish a correct eager FP32 baseline.
2. Tune microbatch and target batch.
3. Benchmark workers and prefetch.
4. Enable a supported precision mode.
5. Verify numerical behavior.
6. Benchmark fused AdamW on CUDA.
7. Add gradient checkpointing only when memory pressure requires it.
8. Add `torch.compile` last and verify eager fallback behavior.

Measurement cautions

Speedtronic logs cumulative counters but computes rates from the start of the current `fit()` invocation. It measures wall time including synchronous Hub work and CPU loss transfer, but does not separate forward, backward, optimizer, checkpoint, or logging costs.

Related: Precision, Data, and Architecture.

Checkpoint and resume

Configure checkpoints

```yaml
checkpoint:
  enabled: true
  directory: checkpoints
  every_steps: 100
  keep_last: 3
  resume: false
```

A relative directory is resolved under `run.output_dir`:

```
<run.output_dir>/checkpoints
```

Request resume

CLI:

```
speedtronic train \
  --config run.yaml \
  --output-dir runs/experiment \
  --resume
```

Python:

```python
from speedtronic.runtime import train_from_config

result = train_from_config(config, resume=True)
```

Configuration:

```yaml
checkpoint:
  resume: true
```

What is restored

State	Restored immediately	Notes
Model parameters/buffers	Yes	Strict `load_state_dict`
AdamW state	Yes	Must match current optimizer structure
Scheduler state	Yes	`last_epoch` restored
Global step	Yes	Controls remaining work
Samples/tokens	Yes	Cumulative telemetry restored
CUDA GradScaler	Deferred	Loaded after `fit()` creates scaler
Coordinator	Deferred	Loaded after coordinator `start()`
Python/NumPy/Torch RNG	Yes	Best effort per subsystem
CUDA RNG	Yes when present	Best effort
DataLoader position	No	Iterator restarts
Epoch/sampler state	No	Not stored
Model train/eval mode	No	Module state remains as constructed

Absolute target behavior

```
target_step = max(self.step, requested_target)
```

Examples:

Restored step	Requested target	New updates
0	10	10
4	10	6
10	10	0
12	10	0

A resumed `TrainResult.steps` is the absolute final step.

Two-step resume demonstration

```
speedtronic train \
  --config configs/smoke.yaml \
  --device cpu \
  --output-dir runs/resume-demo \
  --max-steps 2

speedtronic train \
  --config configs/smoke.yaml \
  --device cpu \
  --output-dir runs/resume-demo \
  --max-steps 4 \
  --resume
```

The CLI updates both `run.max_steps` and `scheduler.max_steps` for each invocation.

Final-step gap

Saving occurs only at exact multiples of `every_steps`. A final off-interval step is not automatically saved.

Final step	Interval	Latest saved
3	2	2
4	2	4
5	2	4

Pointer behavior

`latest.json` points to a checkpoint file. If the pointer is missing or invalid, the manager scans filenames and chooses the highest step.

Crash windows and limitations:

- a higher checkpoint can exist while the pointer is stale;
- a valid pointer to a corrupt file does not fall back to an older one;
- atomic rename does not guarantee power-loss durability without `fsync`;
- temporary and data-file operations assume a local filesystem supporting atomic replacement.

Retention

`keep_last: 3` prunes old `step_*.pt` files after a successful save. The pointer is not pruned.

Use `keep_last: null` for unlimited local retention, with disk-capacity planning.

Verify a checkpoint

```python
import torch

state = torch.load(
    "runs/resume-demo/checkpoints/step_000000000002.pt",
    map_location="cpu",
    weights_only=False,
)

print(state.keys())
print(state["step"], state["samples"], state["tokens"])
```

Only inspect trusted files; `.pt` loading is pickle-capable.

Distributed checkpoints

A distributed trainer checkpoint can include:

- local model and optimizer;
- coordinator counters/baseline;
- pending-delta fields;
- master global state;
- processed-delta set;
- outer momentum.

This duplicates large tensors and increases checkpoint size and memory pressure.

Resume limitations

- DataLoader position is not exact.
- Optimizer updates skipped by a GradScaler overflow still advance the trainer's global step.
- The current coordinator resume ordering can install a fresh global model and then overwrite its baseline with checkpointed local state.
- A worker restart discards partial inner-loop progress unless captured by a later successful global installation path.
- Current config/model compatibility is not validated.

Recovery checklist

1. Confirm the output directory is unchanged.
2. Inspect `latest.json`.
3. Load the selected file in a trusted environment.
4. Verify stored step and counters.
5. Use a target greater than the stored step.
6. Expect possible data repetition after restore.
7. Keep the previous checkpoint until the resumed run is validated.

Related: Checkpoint API, CLI, and DumbDiLoCo recovery.

Observability

Built-in text output

Every event is written to stdout by default:

```
train_start start_step=0 target_step=100 ...
train_step step=1 loss=2.31 ...
checkpoint step=50 path=...
train_end steps=100 elapsed_s=12.3 loss=1.84
```

The master distributed loop can also emit `outer_step` events.

Text file

```yaml
logging:
  level: INFO
  file: runs/demo/train.log
  every_steps: 10
```

Relative logging paths are resolved from the process working directory, not `run.output_dir`.

JSONL file

```yaml
logging:
  json_file: runs/demo/metrics.jsonl
  every_steps: 1
```

Each line is a self-contained JSON event.

Cadence

`logging.every_steps` filters `train_step` events before text, JSONL, and hook dispatch. Lifecycle, checkpoint, compile, and outer events are not filtered by this field.

The trainer still retains every metric dictionary in `TrainResult.metrics`, so long runs can accumulate memory even when output cadence is sparse.

Custom callable hook

```
events = []


def collect(event: str, payload: dict) -> None:
    events.append((event, dict(payload)))


trainer, _ = build_runtime(config)
trainer.logger.hooks.append(collect)
result = trainer.fit()
```

A hook exception is caught and logged; it does not stop training.

Object hook

```
class JsonlHook:
    def on_event(self, event, payload):
        ...
```

`MetricLogger` prefers `hook.on_event` when present and otherwise calls the object directly.

W&B

Install:

```
pip install 'speedtronic[logging]'
```

Attach:

```
from speedtronic.integrations import WandbHook

trainer, _ = build_runtime(config)
wandb_hook = WandbHook(project="speedtronic")
trainer.logger.hooks.append(wandb_hook)
result = trainer.fit()
```

Every emitted event calls:

```
wandb.log({**payload, "event": event})
```

The adapter currently has no public `close()`/finish method.

TensorBoard

```
from speedtronic.integrations import TensorboardHook

tb_hook = TensorboardHook(log_dir="runs/tensorboard")
trainer.logger.hooks.append(tb_hook)
result = trainer.fit()
tb_hook.close()
```

Numeric values are written to `<event>/<key>`.

Direct logger construction

```
from speedtronic.profiling import MetricLogger

logger = MetricLogger(
    level="INFO",
    file="train.log",
    json_file="metrics.jsonl",
    every_steps=5,
    hooks=[collect],
)
```

Attach the logger to a directly constructed `Trainer`.

YAML limitation

`LoggingConfig` has no `hooks` list. Adding one to YAML raises `ConfigError`. Programmatic attachment is required in 2.0.0.

Memory metrics

`MetricLogger` adds `memory_bytes` from the current CUDA or MPS device when available. It does not accept a trainer device parameter, so a non-current CUDA device can report the wrong allocation.

Distributed events

The master emits:

```
{
  "event": "outer_step",
  "outer_step": 4,
  "deltas_found": 3,
  "deltas_included": 2
}
```

An empty round still emits the event with zero included deltas. Because the outer loop runs on a background thread, hook implementations should be thread-safe.

Operational recommendations

- Close file handlers explicitly for long-lived processes.
- Close TensorBoard writers explicitly.
- Finish W&B runs explicitly.
- Keep hooks cheap and nonblocking.
- Use JSONL for durable machine-readable telemetry.
- Monitor hook warnings; failures are intentionally isolated.
- Avoid placing secrets in metric payloads.

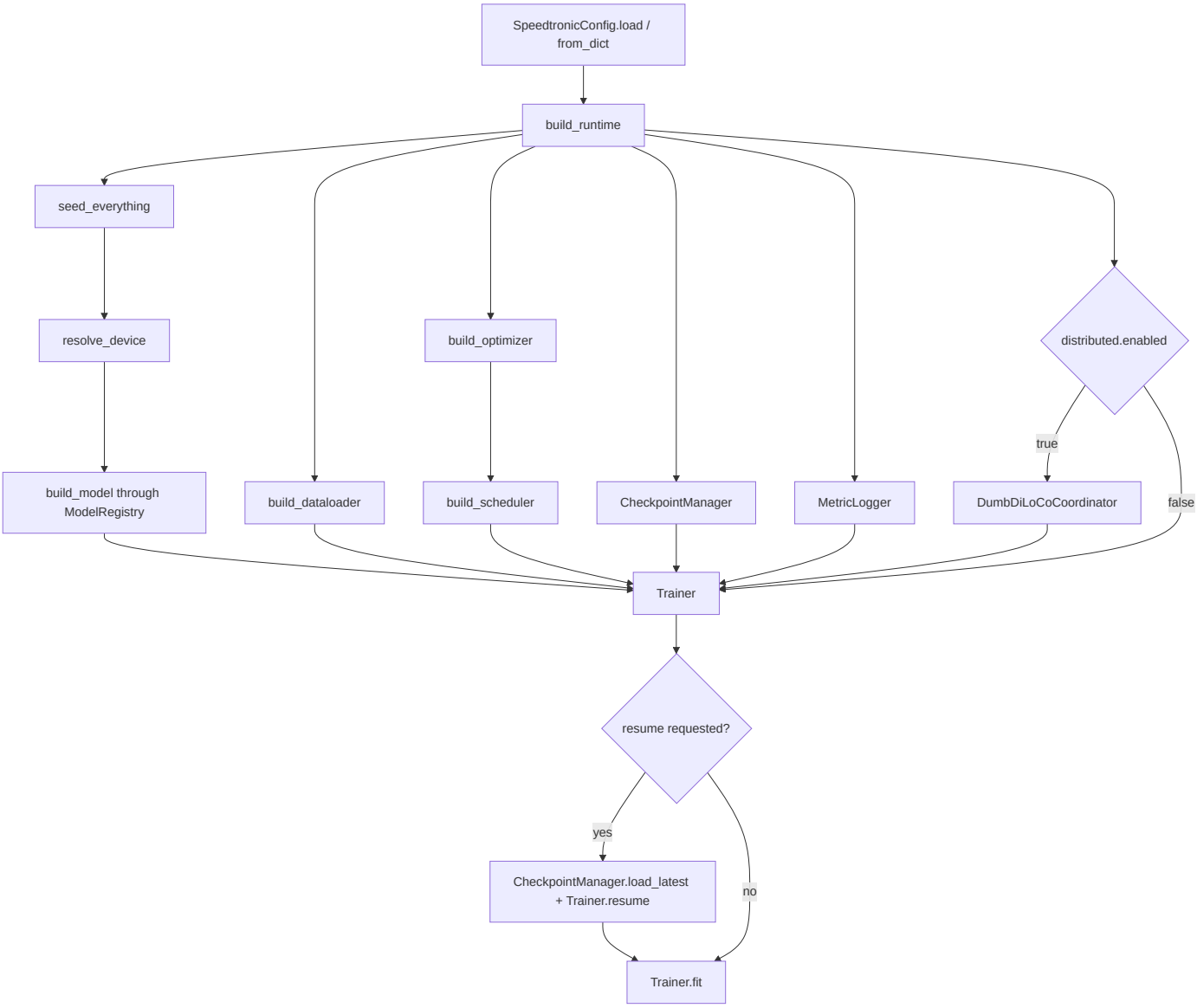
Related: Observability API and DumbDiLoCo.

System architecture

Architectural goal

Speedtronic separates the training engine from a reference architecture. `Trainer` consumes a PyTorch module plus a small batch/output protocol; model construction is delegated to a registry. The bundled GPT-style model is an example and convenience, not a dependency of the loop.

Composition root

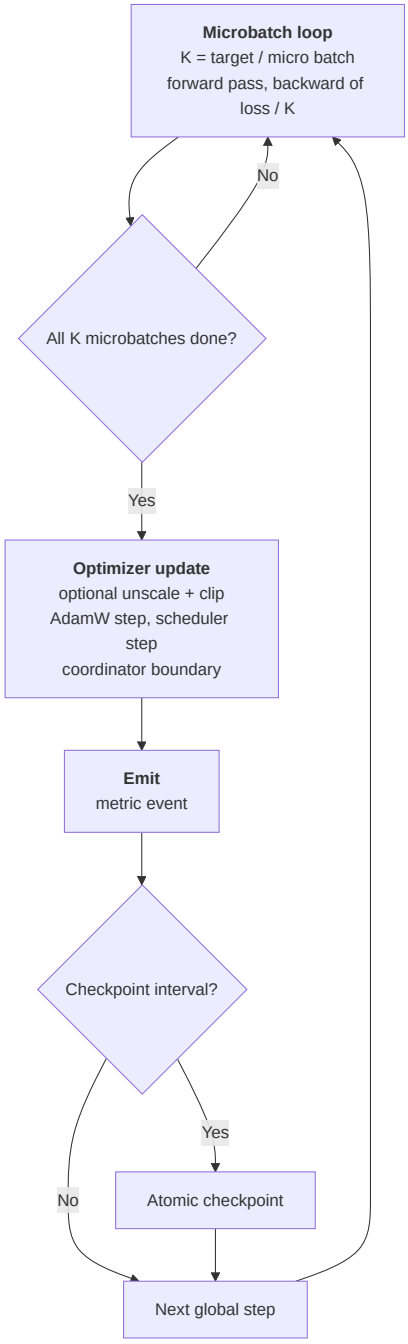


`build_runtime()` returns both the `Trainer` and `CheckpointManager`. `train_from_config()` keeps only the trainer and immediately calls `fit()`.

Runtime construction order

1. Convert a dictionary to `SpeedtronicConfig`; validate any config object.
2. Seed Python, NumPy, PyTorch CPU, and all CUDA devices.
3. Resolve `auto`, CPU, CUDA, or MPS.
4. Build the model through the process-global registry.
5. Build the data source and DataLoader.
6. Build AdamW, optionally probing fused mode.
7. Build a `LambdaLR` warmup/cosine or warmup/constant schedule.
8. Resolve checkpoint paths under `run.output_dir`.
9. Build the text/JSONL `MetricLogger`.
10. Construct the DumbDiLoCo coordinator when enabled.
11. Compute the effective global step target.
12. Construct the `Trainer`.
13. Load the latest local checkpoint when resume was requested.

One global optimizer step



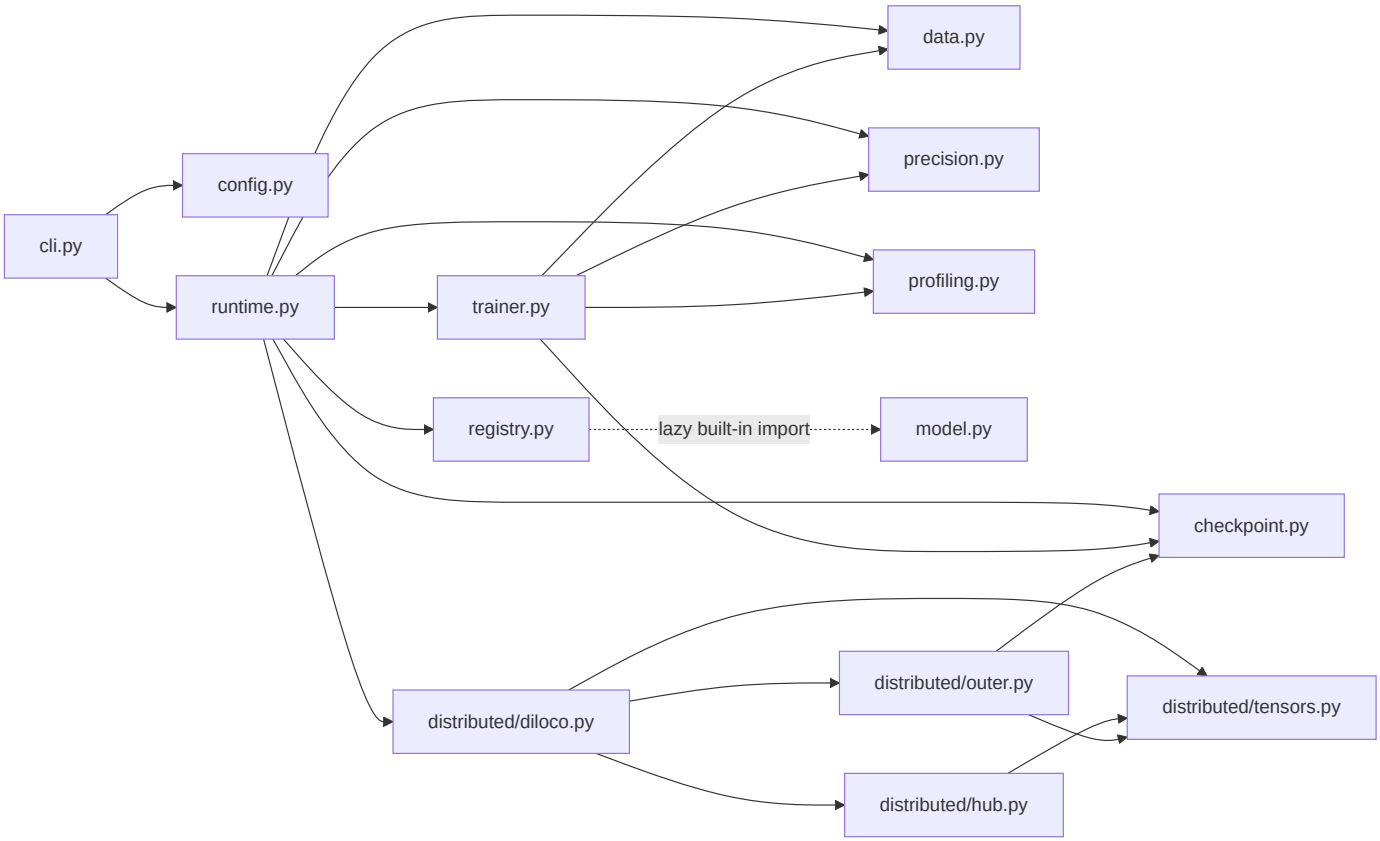
Ordering guarantees

Within a global step, Speedtronic performs:

1. Complete forward/backward work for every microbatch.
2. One optimizer update.
3. One scheduler update.
4. One coordinator callback.

5. Optional optimizer-state reset.
6. One metric record.
7. Optional checkpoint write.

Module dependency map



Feature boundaries

Architecture neutrality

The loop is architecture-neutral when the model follows its contracts. Configuration and factory argument passing are more transformer-oriented: `ModelConfig` carries vocabulary, context, layer, head, width, FFN, dropout, tying, and RoPE fields, and `build_model()` always passes those fields.

Local versus distributed

`Trainer` has no knowledge of Hub details. It depends only on `SyncCoordinator`:

```python
class SyncCoordinator(Protocol):
    def start(self) -> None: ...
    def after_optimizer_step(self, model: Any, step: int) -> bool | None: ...
    def stop(self) -> None: ...
    def state_dict(self) -> dict[str, Any] | None: ...
    def load_state_dict(self, state: dict[str, Any]) -> None: ...
```

A truthy callback result asks the trainer to clear inner optimizer state when `distributed.reset_inner_optimizer` is enabled.

Performance fallbacks

Capability-oriented features degrade rather than terminate local training:

- unsupported fused AdamW construction falls back to regular AdamW;
- `torch.compile` construction or runtime failure falls back to eager execution;
- missing/failing gradient-checkpointing hooks log a warning;
- Hub failures log and retry without terminating local training;
- unsupported explicit mixed precision generally falls back to FP32, with the CPU BF16 caveat documented on the precision page.

Failure boundaries

Failure	Policy
Invalid configuration key/value	Raise `ConfigError`
Model/data/forward error	Propagate and stop the run
Checkpoint I/O error	Propagate
Metric hook error	Log warning and continue
Compile failure	Disable compiled path and retry eagerly
Fused AdamW failure	Use regular AdamW
Hub transient/permanent error	Retry, log, and usually continue local training
Resume with no checkpoint	Warn and start without restored state

v2 additions

- `optimizers.py` supplies role-aware Muon/AdamW routing, Newton–Schulz, Muon+, and cautious wrapping.
- `shapes.py` validates effective batch/model dimensions before optimization.
- `scheduling.py` assigns disjoint CUDA stage streams and falls back to sequential backward on CPU/MPS.
- DumbDiLoCo now dispatches Hub work through bounded background I/O lanes.

See the v2 overview for the complete composition and compatibility notes.

Source ownership

Every implementation file is mapped in the generated source inventory. The curated source coverage page explains what that inventory means.

Command-line interface

The package installs one console entry point:

```
speedtronic = speedtronic.cli:main
```

The module entry point delegates to the same function:

```
python -m speedtronic --help
```

`speedtronic train`

```
speedtronic train --config PATH
  [--resume]
  [--device DEVICE]
  [--max-steps N]
  [--output-dir PATH]
```

Option	Required	Behavior
`--config`	Yes	YAML or JSON configuration path
`--resume`	No	Load the latest local checkpoint if one exists
`--device`	No	Override `run.device`; commonly `auto`, `cpu`, `cuda`, or `mps`
`--max-steps`	No	Override the absolute local optimizer-step target and scheduler horizon
`--output-dir`	No	Override `run.output_dir` before runtime composition

After completion, the CLI prints:

```
completed steps=<global-step> samples=<cumulative-samples> tokens=<cumulative-tokens> loss=<value-or-
None>
```

The counters are cumulative and include restored state.

Override order

CLI processing occurs after configuration construction:

1. Load and validate the file.
2. Mutate `run.output_dir` when `--output-dir` is present.
3. Mutate both `run.max_steps` and `scheduler.max_steps` when `--max-steps` is present.
4. Pass `device` and `max_steps` again as runtime keyword overrides.
5. Load the checkpoint from the resolved output directory when resume is active.

Because checkpoint paths are constructed after output override, resume searches the overridden output directory.

`speedtronic validate`

```
speedtronic validate --config PATH [--format {yaml,json}]
```

Validation:

- loads YAML or JSON;
- rejects unknown mapping keys;
- applies defaults and section validation;
- prints the normalized configuration;
- does not import the runtime eagerly.

It does not:

- build or register a model;
- open a data file;
- load a serialized dataset;
- resolve the requested device;
- check hardware precision support;
- perform a model forward;
- connect to Hugging Face Hub.

> ⚠ **SECRET EXPOSURE**
>
> `validate` calls `to_dict()` and `to_yaml()` without secret redaction. A token placed under `distributed.token` can be printed. Prefer `HF_TOKEN`/SDK authentication and avoid validating a file that contains a secret.

Exit behavior

Outcome	Exit code
Success	`0`
Handled configuration, file, key, runtime, type, or value error	`2`

The explicit exception tuple does not cover every possible lower-level `OSError` or import failure.

Examples

Four CPU steps

```
speedtronic train --config configs/smoke.yaml --device cpu
```

Resume an existing target

```
speedtronic train --config configs/smoke.yaml --device cpu --resume
```

Override output and target

```
speedtronic train \
  --config configs/smoke.yaml \
  --output-dir runs/experiment-42 \
  --max-steps 100
```

Validate as JSON

```
speedtronic validate --config configs/smoke.yaml --format json
```

Programmatic equivalent

```
from speedtronic import SpeedtronicConfig
from speedtronic.runtime import train_from_config

config = SpeedtronicConfig.load("configs/smoke.yaml")
result = train_from_config(
    config,
    resume=True,
    device="cpu",
    max_steps=100,
)
```

`train_from_config()` accepts a config object or dictionary and returns `TrainResult`; see Runtime and Trainer.

Configuration reference

Speedtronic configuration is implemented by mutable dataclasses in `speedtronic.config`. Each section validates selected invariants in `__post_init__`; the aggregate applies cross-section defaults.

Accepted forms

```
from speedtronic import SpeedtronicConfig, load_config

config = SpeedtronicConfig.from_dict({...})
config = SpeedtronicConfig.from_yaml("run.yaml")
config = SpeedtronicConfig.load("run.json")
config = SpeedtronicConfig.load("name: demo\nmax_steps: 10")
config = load_config({"max_steps": 10})
```

`load()` heuristically distinguishes inline YAML/JSON from paths. Unknown root and nested mapping keys raise `ConfigError`.

Root sections

```
run: {}
model: {}
data: {}
optimizer: {}
scheduler: {}
precision: {}
checkpoint: {}
logging: {}
distributed: {}
gradient_checkpointing: false
compile: false
```

`RunConfig`

Field	Type	Default	Meaning and validation
`name`	`str`	`speedtronic-run`	Descriptive run name; not used as a filesystem path
`seed`	`int`	`1234`	Seeds Python, NumPy when available, PyTorch CPU, and CUDA
`device`	`str`	`auto`	Requested device; `auto` selects CUDA, MPS, then CPU
`max_steps`	`int`	`1000`	Positive absolute local optimizer-step target
`output_dir`	`str`	`runs`	Base for relative checkpoint and distributed state paths
`log_every`	`int | None`	`null`	Positive; copied to logging cadence only when logging cadence remains `1`

`ModelConfig`

Field	Type	Default	Meaning and validation
`name`	`str`	`reference_transformer`	Registry key; membership is not checked during config parsing
`vocab_size`	`int`	`512`	Positive vocabulary size
`max_seq_len`	`int`	`128`	Positive reference-model context limit; mapped to `block_size`
`n_layer`	`int`	`4`	Positive layer count
`n_head`	`int`	`8`	Positive query-head count
`n_kv_head`	`int | None`	`null`	Resolves to `n_head`; positive and divides `n_head`
`d_model`	`int`	`256`	Positive hidden width; divisible by `n_head`
`d_ff`	`int | None`	`null`	Resolves to `4 * d_model`; positive
`dropout`	`float`	`0.0`	Must satisfy `0 <= dropout < 1`
`tie_weights`	`bool`	`true`	Reference-model embedding/output weight tying
`rope_base`	`float`	`10000.0`	Rotary base; no positivity check in `ModelConfig`
`gradient_checkpointing`	`bool`	`false`	Synchronized with root-level flag

These fields are reference-transformer-oriented even though the trainer itself is architecture-neutral.

`DataConfig`

Field	Type	Default	Meaning and validation
`text_path`	`str | null`	`null`	UTF-8 text file; selected before serialized/synthetic data
`dataset`	`str | null`	`null`	Serialized dataset path in YAML; may hold a Dataset object programmatically
`synthetic`	`bool`	`true`	Enables synthetic fallback
`num_tokens`	`int`	`100000`	Positive; synthetic implementation uses it as **sample count**
`vocab_size`	`int | null`	`null`	Positive if explicit; aggregate normally fills from model
`block_size`	`int | null`	`null`	Positive after aggregate initialization; normally model context length
`micro_batch_size`	`int`	`1`	Positive loader batch size
`target_batch_size`	`int`	`1`	Positive, at least microbatch, and divisible by microbatch
`num_workers`	`int`	`0`	Non-negative DataLoader worker count
`prefetch_factor`	`int | null`	`null`	Positive if explicit; defaults to 2 when workers are enabled
`pin_memory`	`bool | null`	`null`	Explicit override or runtime device-derived default
`shuffle`	`bool`	`false`	Applied to map-style datasets; forced off for iterable datasets
`drop_last`	`bool`	`true`	Drop incomplete loader batches
`seed`	`int | null`	`null`	Aggregate fills from `run.seed`
`max_steps`	`int | null`	`null`	Positive; used as local target when `run` target is not explicitly overridden

Derived accumulation:

```
config.data.accumulation_steps
# target_batch_size // micro_batch_size
```

⚠ `num_tokens` **NAMING**

> For synthetic data, `num_tokens=256` creates 256 samples, each containing `block_size` input tokens and one next-token target. It is not a literal total-token budget.

`OptimizerConfig`

Field	Type	Default	Meaning and validation
`name`	`str`	`adamw`	Only `adamw` is supported
`lr`	`float`	`0.0003`	Positive learning rate
`betas`	`tuple[float, float]`	`(0.9, 0.95)`	Exactly two values in `[0, 1)`
`eps`	`float`	`1e-8`	Positive Adam epsilon
`weight_decay`	`float`	`0.1`	Non-negative
`fused`	`bool | null`	`null`	Auto-probe, force, or disable fused AdamW
`grad_clip`	`float | null`	`null`	Positive global gradient-norm limit

Fused construction is best-effort. Failure falls back to regular AdamW.

`SchedulerConfig`

Field	Type	Default	Meaning and validation
`name`	`str`	`cosine`	`cosine` or `constant`
`warmup_steps`	`int`	`100`	Non-negative optimizer-step warmup
`max_steps`	`int`	`1000`	Positive schedule horizon
`min_lr_ratio`	`float`	`0.1`	Cosine floor in `[0, 1]`; ignored by constant schedule

The warmup factor is `(step + 1) / warmup_steps`, floored at `1e-8`. The cosine factor clamps progress to `[0, 1]`.

> **⚠ SCHEDULE TARGET MISMATCH**

> `SchedulerConfig` rejects `max_steps <= 0`, making the aggregate branch that attempts to infer it from `run.max_steps` unreachable. If a Python config omits `scheduler.max_steps`, it remains 1000 even when `run.max_steps` is 10. The CLI `--max-steps` override updates both fields; direct Python overrides do not.

`PrecisionConfig`

Field	Type	Default	Meaning and validation
`mode`	`str`	`auto`	`auto`, `bf16`, `fp16`, or `fp32`
`dtype`	`str | null`	`null`	`bf16`, `fp16`, or `fp32`; affects resolution only when mode is `auto`

A string section is accepted as shorthand:

```
precision: bf16
```

Conflicting values such as `mode: fp32` and `dtype: bf16` are accepted; the explicit mode wins and `dtype` is ignored.

`CheckpointConfig`

Field	Type	Default	Meaning and validation
`enabled`	`bool`	`true`	Enable interval saves from the trainer
`directory`	`str`	`checkpoints`	Relative paths are rooted under `run.output_dir`
`every_steps`	`int`	`500`	Positive global-step interval
`keep_last`	`int | null`	`3`	Positive retained count or `null` for unlimited
`resume`	`bool`	`false`	Request local checkpoint loading during runtime construction

Saving happens only at exact interval boundaries. The trainer does not force a final checkpoint when the final step is off interval.

`LoggingConfig`

Field	Type	Default	Meaning and validation
`level`	`str`	`INFO`	Uppercased; unknown names map to INFO in the logger
`file`	`str | null`	`null`	Optional text output; relative to process working directory
`every_steps`	`int`	`1`	Positive cadence for `train_step` text, JSONL, and hook delivery
`json_file`	`str | null`	`null`	Optional JSONL output; relative to process working directory

There is no YAML `hooks` field. W&B, TensorBoard, and custom hooks require programmatic logger construction or mutation.

`DistributedConfig`

Field	Type	Default	Meaning and validation
`enabled`	`bool`	`false`	Construct `DumbDiLoCoCoordinator`
`mode`	`str`	`dumb_diloco`	`dumb_diloco` or disabled `single`
`role`	`str`	`single`	`single`, `master`, or `worker`
`node_id`	`str | null`	`null`	Explicit path-safe unique ID
`collaborators`	`list[str]`	`[]`	Master attempts to grant each user `write`
`inner_steps`	`int`	`500`	Positive local steps between uploads
`poll_interval`	`float`	`60.0`	Positive worker/master polling interval
`outer_lr`	`float`	`0.7`	Positive Nesterov outer learning rate
`outer_momentum`	`float`	`0.9`	Value in `[0, 1)`
`repo_id`	`str | null`	`null`	Required for enabled distributed mode
`token`	`str | null`	`null`	Optional Hub token; prefer SDK environment authentication
`cache_dir`	`str`	`.speedtronic/hub`	Hub cache, relative to process working directory
`state_dir`	`str`	`.speedtronic/diloco`	Relative state root under `run.output_dir`, then `/<node_id>`
`retry_initial`	`float`	`1.0`	Positive first retry delay
`retry_max`	`float`	`60.0`	At least `retry_initial`
`retry_attempts`	`int`	`6`	Positive total attempts per Hub operation
`reset_inner_optimizer`	`bool`	`true`	Clear optimizer state after successful upload/load callback

When enabled with `role: single`, the role becomes `master` because a repository is required. The code treats this as a master-by-default convenience.

Root booleans

```yaml
gradient_checkpointing: true
compile: true
```

`compile` also accepts:

```yaml
compile:
  enabled: true
```

Top-level values pass through `bool(...)`; quoted strings such as `"false"` become truthy. YAML booleans should remain unquoted.

Aggregate normalization

`SpeedtronicConfig.__post_init__` performs:

1. Positive run target check.
2. Distributed role fallback.
3. `data.block_size = model.max_seq_len` when null.
4. `data.vocab_size = model.vocab_size` when null.
5. `data.seed = run.seed` when null.
6. `logging.every_steps = run.log_every` when the former remains 1.
7. Scheduler fallback that is only reachable for a non-positive value.
8. `data.max_steps` scheduler adoption when the scheduler still has its default 1000 horizon.
9. If either gradient-checkpointing flag is true, set both root and model flags true.

Top-level aliases

These keys are moved into `run` before schema validation:

```yaml
name: demo
seed: 42
device: cpu
max_steps: 100
output_dir: runs/demo
log_every: 10
```

If both a top-level alias and nested `run` field exist, the top-level alias overwrites the nested value for that field.

Compatibility section aliases:

```yaml
hub: {}       # accepted only when distributed is absent
diloco: {}    # accepted only when distributed is absent
```

Serialization and redaction

```python
plain = config.to_dict()
safe = config.to_dict(redact_secrets=True)
yaml_text = config.to_yaml(redact_secrets=True)
path = config.save("run.yaml")
```

`save()` always redacts `distributed.token`. `to_dict()` and `to_yaml()` default to **unredacted** output. Checkpoints also request redaction.

Path semantics

Setting	Relative-path base
`checkpoint.directory`	`run.output_dir`
`distributed.state_dir`	`run.output_dir`, then `node_id`
`distributed.cache_dir`	Process working directory
`logging.file`	Process working directory
`logging.json_file`	Process working directory
`data.text_path`	Process working directory
serialized `data.dataset`	Process working directory

v2 optimizer fields

```yaml
optimizer: muon
muon_plus: true
cautious: true
```

The canonical mapping form is:

```yaml
optimizer:
  name: muon          # adamw or muon
  lr: 0.0003
  muon_plus: false
  cautious: false
  muon_momentum: 0.95
  muon_ns_steps: 5
  muon_norm_eps: 0.00000001
```

`optimizer: muon` and the root-level `muon_plus`/`cautious` forms are normalized into `OptimizerConfig`. Conflicting duplicate values are rejected. `muon_plus` requires `name: muon`; `cautious` works with either optimizer.

v2 systems fields

```yaml
shape_validation:
  enabled: true
  alignment: auto
  check_batch: true
  check_sequence: true
  check_model: true
  check_vocab: false
  warn_on_cpu: false

ooo_backprop: false
ooo_streams: 4
```

`ooo_streams` is constrained to 1–8. Shape warnings are non-fatal. See the v2 optimizer, scheduling, and shape validation pages.

v2 distributed fields

```yaml
distributed:
  async_delta_upload: true
  delta_upload_queue_size: 1
  delta_upload_overflow: skip
  delta_upload_shutdown_timeout: 5.0
  async_global_poll: true
```

Queue size is intentionally one in v2. An occupied upload slot causes the next boundary to skip and log rather than block the inner loop. Set both async flags to `false` for the legacy synchronous transport path.

```yaml
run:
  name: local
  seed: 1234
  device: cpu
  max_steps: 4
  output_dir: runs/local

model:
  name: reference_transformer
  vocab_size: 128
  max_seq_len: 32
  n_layer: 2
  n_head: 4
  n_kv_head: 2
  d_model: 64
  d_ff: 128

data:
  synthetic: true
  num_tokens: 32
  block_size: 32
  micro_batch_size: 1
  target_batch_size: 2
  num_workers: 0

optimizer:
  lr: 0.0003
  weight_decay: 0.01

scheduler:
  name: cosine
  warmup_steps: 1
  max_steps: 4

precision:
  mode: fp32

checkpoint:
  enabled: true
  directory: checkpoints
  every_steps: 2
  keep_last: 2

logging:
  level: INFO
  every_steps: 1
```

For loader and model implementation details, continue to Data and Reference model. For execution, see Runtime and Trainer.

Runtime and Trainer API

Runtime functions

`seed_everything(seed: int) -> None`

Seeds:

- Python `random`;
- `PYTHONHASHSEED` in the current process environment;
- NumPy if it can be imported;
- PyTorch CPU;
- all CUDA devices when CUDA is available.

NumPy failures are ignored.

`build_optimizer(model, config, device) -> torch.optim.Optimizer`

Builds AdamW from `config.optimizer`:

```
lr, betas, eps, weight_decay
```

When `optimizer.fused` is null, `supports_fused_adamw(device)` probes CUDA. A forced fused value is also attempted. `TypeError`, `RuntimeError`, or `ValueError` during construction falls back to ordinary AdamW.

`build_scheduler(optimizer, config) -> LambdaLR`

Creates warmup and either cosine or constant decay:

```
warmup: max(1e-8, (step + 1) / warmup_steps)
constant: 1.0
cosine: min_ratio + (1 - min_ratio) * 0.5 * (1 + cos(pi * progress))
```

The scheduler advances once per global optimizer update, not per microbatch.

`build_logger(config) -> MetricLogger`

Maps `logging.level`, `file`, `json_file`, and `every_steps`. It does not attach hooks.

`build_runtime(...)`

```
build_runtime(
    config: SpeedtronicConfig | dict[str, Any],
    *,
    resume: bool = False,
    device: str | torch.device | None = None,
    max_steps: int | None = None,
) -> tuple[Trainer, CheckpointManager]
```

The function validates, seeds, resolves hardware, and constructs every local component. If distributed mode is enabled, it also creates `DumbDiLoCoCoordinator`.

Effective target precedence:

```
explicit build_runtime(max_steps=...)
→ data.max_steps
→ run.max_steps
```

The expression is implemented as `max_steps if provided else (data.max_steps or run.max_steps)`.

Resume lookup happens after the trainer is created. A missing checkpoint is a warning, not an error.

`train_from_config(...) -> TrainResult`

Builds the runtime and immediately calls `Trainer.fit()`.

Aliases:

```
run = train_from_config
train = train_from_config
```

`SyncCoordinator`

```
@runtime_checkable
class SyncCoordinator(Protocol):
    def start(self) -> None: ...
    def after_optimizer_step(self, model: Any, step: int) -> bool | None: ...
    def stop(self) -> None: ...
    def state_dict(self) -> dict[str, Any] | None: ...
    def load_state_dict(self, state: dict[str, Any]) -> None: ...
```

`after_optimizer_step()` may return true to request inner optimizer state reset. DumbDiLoCo returns true after a successful upload **or** global installation.

`TrainResult`

```
@dataclass
class TrainResult:
    steps: int
    samples: int
    tokens: int
    final_loss: float | None
    elapsed_s: float
    metrics: list[dict[str, Any]]
```

Field	Meaning
`steps`	Final absolute global step, including resumed state
`samples`	Cumulative sample counter
`tokens`	Cumulative token counter
`final_loss`	Mean final-update loss for this invocation, or null when no update ran
`elapsed_s`	Duration of this invocation
`metrics`	Every metric record generated by this invocation

`metrics` grows on every step even when the logger suppresses output by cadence.

`Trainer`

```
Trainer(
    model,
    optimizer,
    dataloader,
    *,
    device,
    config=None,
    scheduler=None,
    precision=None,
    logger=None,
    checkpoint_manager=None,
    coordinator=None,
    max_steps=None,
    start_step=0,
)
```

The constructor:

- converts a dictionary config;

- stores the model/optimizer/loader/device;
- creates a default logger;
- resolves a precision plan when omitted;
- establishes cumulative counters;
- configures gradient checkpointing;
- optionally constructs `torch.compile(model)`.

Compatibility aliases:

```
TrainingEngine = Trainer
SpeedtronicTrainer = Trainer
```

Batch movement and counting

`_move_batch()` recursively transfers tensors in dictionaries, tuples, and lists to the selected device with `non_blocking=True`.

`_batch_size_and_tokens()` chooses:

- dictionary `input_ids`, then `inputs`; or
- the first tuple/list field.

For a multi-dimensional dictionary input with `attention_mask`, tokens equal the mask sum. Unsupported values report one sample and one token.

Loss resolution

Mapping output

- `{"loss": tensor}` uses the loss directly.
- `{"logits": 3d_tensor}` without loss triggers trainer-side causal cross-entropy.

Tensor output

A bare tensor is a direct loss, even if it is 3-D.

Tuple/list output

- A scalar first tensor is a direct loss.
- A 3-D first tensor is treated as causal logits when labels have compatible shapes.

Inferred causal loss

For logits `(B, T, V)`:

- same-length labels are treated as already next-token aligned and use the full logit row;
- logits are truncated to `logits[:, :-1]` only for `(B, T-1)` labels;
- `-100` is ignored;
- all-ignored labels produce a differentiable zero tied to the logits.

The bundled datasets and the bundled model share this one-label-per-position contract, so the first target is not skipped.

Accumulation and optimization

For `K = accumulation_steps`:

1. Run each of K microbatches.
2. Mean any non-scalar loss.
3. Backpropagate `loss / K` immediately.
4. On the final microbatch, optionally unscale and clip.
5. Perform one optimizer update.
6. Clear gradients with `set_to_none=True`.

The reported loss is the arithmetic mean of the K unscaled microbatch losses.

Feature setup

Gradient checkpointing

If enabled, the trainer calls:

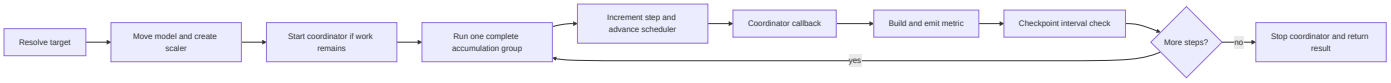
```
model.set_gradient_checkpointing(True)
```

A missing hook, attribute error, or any hook exception logs a warning and continues.

Compilation

`torch.compile` construction failure leaves the eager model active. Any exception from a compiled forward disables compilation and reruns that batch eagerly. The fallback is broad: it can hide a non-compilation model exception if eager execution succeeds.

Global-step lifecycle



The target is `max(self.step, requested_target)`. `fit()` always calls coordinator `stop()` in `finally`, even on failure.

`fit(max_steps=None) -> TrainResult`

1. Validate positive target.
2. Move model to the device.
3. Select original or compiled model.
4. Create a new `GradScaler` and restore deferred scaler state.
5. Zero gradients.
6. Capture invocation start time/counters.
7. Wrap loader in `infinite_batches`.
8. Start coordinator if needed.
9. Apply deferred coordinator checkpoint state after startup.
10. Emit `train_start`.
11. Run complete global steps.
12. Emit `train_end` and return.

`Trainer.train` is an alias for `fit`.

Resume

`Trainer.resume(state)` immediately restores:

- model state;
- optimizer state;
- scheduler state;
- global step/sample/token counters;
- RNG state.

Scaler and coordinator state are staged for `fit()`; v2 coordinators can apply prepared baseline state during startup before any remote refresh.

Saved precision and config copies are informational; they are not compared with or applied to the current runtime.

Per-step metrics

Key	Meaning
`step`	New cumulative step
`loss`	Mean microbatch loss
`lr`	First parameter-group LR after scheduler advancement
`steps_per_sec`	Invocation-local step rate
`samples_per_sec`	Invocation-local sample rate
`tokens_per_sec`	Invocation-local token rate
`samples`	Cumulative samples
`tokens`	Cumulative tokens
`micro_batches`	Microbatches in this update

The reported LR is one scheduler position ahead of the LR that produced the just-completed update.

v2 integration

`build_runtime()` resolves precision, validates startup shapes, builds AdamW or hybrid Muon, and passes a resolved `PrecisionPlan` to the trainer. The trainer starts the opt-in conservative CUDA stage scheduler when `ooo_backprop` is enabled and removes its hooks when the fit ends.

See v2 optimizers, out-of-order backprop, and shape validation.

Important caveats

- A `GradScaler` may skip an overflowing optimizer update; the trainer emits an event and keeps the global step unchanged for that attempt.
- The trainer explicitly switches the model to `train()` mode before fitting.
- `fit()` creates a new scaler each invocation.
- A stopped v2 coordinator resets its started flag; a later `fit()` can start fresh background lanes, subject to the bounded shutdown timeout.
- A compile runtime failure reruns the same batch eagerly, which can duplicate side effects.

- Every per-microbatch loss is copied to CPU, synchronizing CUDA on each microbatch.

The generated source inventory lists all private methods and exact AST-derived signatures.

Data pipeline API

Source-selection order

`build_dataloader()` selects the first available source:

1. Explicit `dataset=` argument.
2. `data.text_path`.
3. A live Dataset object stored in `data.dataset` when called programmatically.
4. A serialized dataset path.
5. Synthetic data when `synthetic=true`.
6. Error when synthetic data is disabled and no source exists.

`CharTokenizer`

```
CharTokenizer(vocab_size: int = 256)
```

UTF-8 byte tokenizer:

```
[byte % vocab_size for byte in text.encode("utf-8")]
```

It is deterministic and stream-friendly but not reversible when `vocab_size < 256`.

Methods:

- `encode(text) -> list[int]`
- `__call__(text) -> list[int]`

`SyntheticTokenDataset`

```
SyntheticTokenDataset(
    num_samples=10_000,
    block_size=128,
    vocab_size=512,
    seed=1234,
)
```

A map-style dataset that returns:

```
{
    "input_ids": tokens[:-1],
    "labels": tokens[1:],
}
```

Each index uses a generator seeded from the dataset seed plus the index, making item contents independent of access order. Negative indexes are normalized from the end.

`DataConfig.num_tokens` is passed as `num_samples` by the runtime, despite its name.

`TextFileTokenDataset`

```
TextFileTokenDataset(
    path,
    block_size,
    tokenizer=None,
    vocab_size=256,
)
```

An `IterableDataset` that reads UTF-8 text in chunks of `block_size * 4` characters. It combines tokenized chunks with a carry, emits groups of `block_size + 1` tokens, and retains the remainder. A final carry of at least two tokens is emitted as a short block.

The default tokenizer is `CharTokenizer`.

Tokenizer contract

A tokenizer may expose:

- `encode(text)`, or
- `__call__(text)`.

Tensor-like results are converted through `.tolist()`; every token is cast to `int`.

> **⚠ BOUNDARY-SENSITIVE TOKENIZERS**
>
> The stream carries token IDs, not raw text. This is safe for the bundled byte tokenizer but can split merges or normalization-sensitive BPE/SentencePiece tokens at read boundaries.

> **⚠ ITERABLE WORKERS**
>
> `TextFileTokenDataset` does not shard by DataLoader worker. With `num_workers > 1`, every worker can read the same file. Implement a worker-aware iterable dataset for parallel production text ingestion.

`collate_causal(batch)`

Pads a list of causal dictionaries to the longest input:

Field	Padding
`input_ids`	Integer zero
`labels`	Integer `-100`
`attention_mask`	Boolean false

Every item must contain `input_ids` and `labels` with compatible shapes.

`collate_batch(batch)`

First item	Result
`dict`	Delegate to `collate_causal`
`tuple` or `list`	Transpose equal-width items; stack equal-shaped tensor columns
`torch.Tensor`	Stack tensors
Other	Return the original list

Empty batches raise `ValueError`. Tuple/list item widths must match.

The function is not a universal PyTorch collator: arbitrary dictionaries are interpreted as causal language-model records.

`infinite_batches(loader)`

Returns an iterator that repeatedly traverses the loader. If one complete pass yields no batches, it raises `RuntimeError("data loader produced no batches")`.

This makes global-step training independent of finite dataset length.

`build_dataloader(...)`

```
build_dataloader(
    config,
    *,
    dataset=None,
    tokenizer=None,
    pin_memory_device=False,
    vocab_size=None,
) -> DataLoader
```

Dataset construction

- Serialized datasets load through `torch.load(..., weights_only=False)` and must contain a PyTorch Dataset.
- Synthetic data receives `block_size`, explicit/runtime `vocab_size`, and `data.seed`.

Loader arguments

Argument	Behavior
`batch_size`	`data.micro_batch_size`
`shuffle`	`data.shuffle`, forced false for iterable datasets
`num_workers`	`data.num_workers`
`pin_memory`	Explicit `data.pin_memory`, otherwise `pin_memory_device`
`drop_last`	`data.drop_last`
`collate_fn`	`collate_batch`
`prefetch_factor`	`data.prefetch_factor` or 2 when workers enabled
`persistent_workers`	True when workers enabled

Precedence and vocab caveats

The explicit `vocab_size` argument wins. `build_runtime()` always passes `model.vocab_size`, so a separately configured `data.vocab_size` has no effect through the standard YAML path.

Programmatic custom dataset

```
from speedtronic.data import build_dataloader

loader = build_dataloader(
    config.data,
    dataset=my_dataset,
    tokenizer=my_tokenizer,
    pin_memory_device=False,
    vocab_size=config.model.vocab_size,
)
```

To use the custom loader through the config-only runtime, inject it or construct `Trainer` directly; `build_runtime()` does not expose dataset/tokenizer override parameters.

Token and sample accounting

Trainer token counts are based on input tensor shape, not tokenizer semantics. A dictionary input with an attention mask uses the mask sum. This can differ from a tokenizer's reported token count after normalization or special tokens.

Text configuration

```
data:
  text_path: /absolute/path/to/train.txt
  block_size: 128
  micro_batch_size: 2
  target_batch_size: 8
  num_workers: 0
  prefetch_factor: null
  pin_memory: null
  shuffle: false
  drop_last: true
```

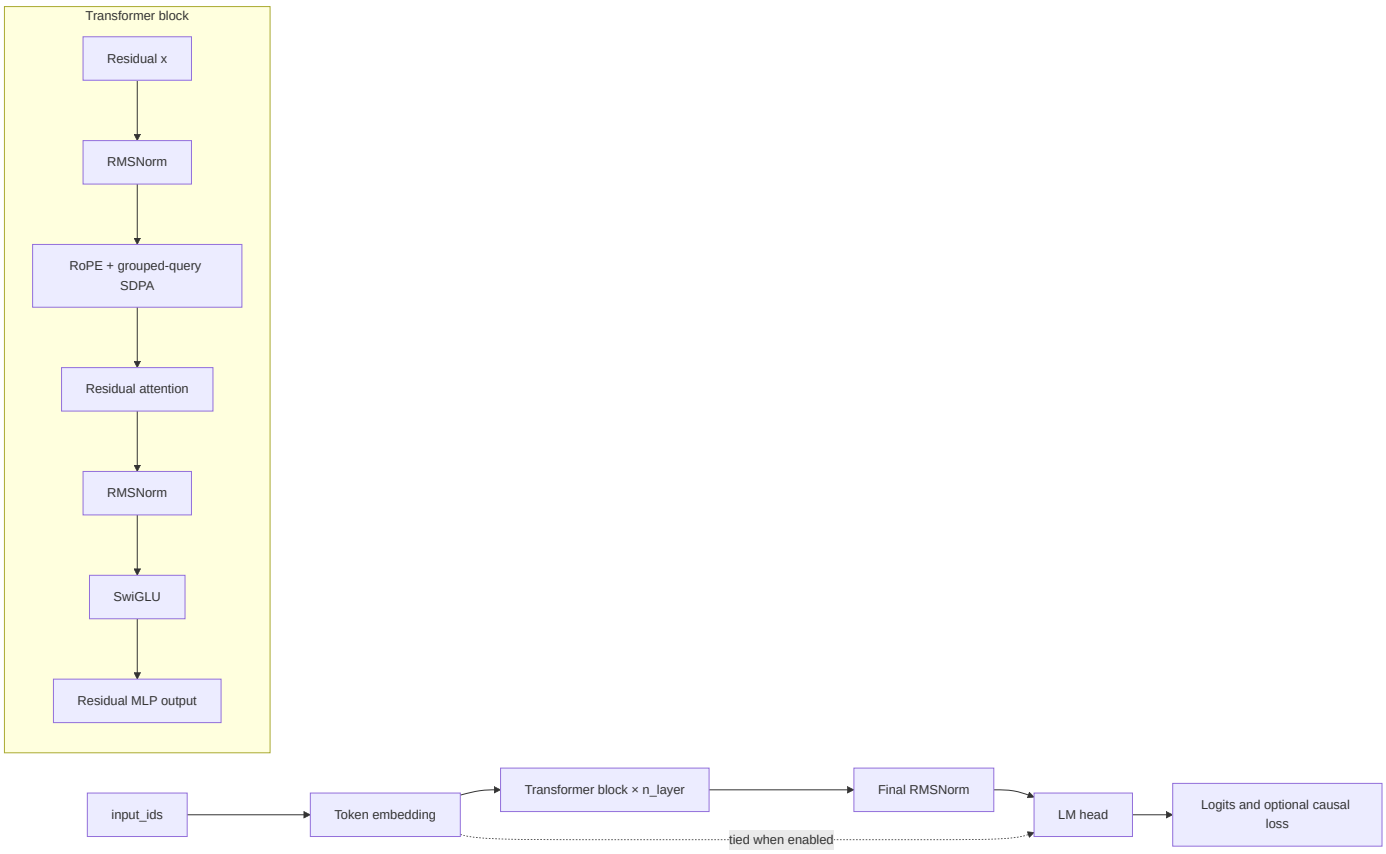
Use `num_workers: 0` with the bundled text dataset unless you provide a sharding implementation.

For the trainer-facing contracts, continue with Runtime and Trainer and Custom data.

Reference transformer

The bundled model is a compact decoder-only language model. It demonstrates the registry and trainer contracts but is not a requirement of the engine.

Architecture



`GPTConfig`

```
GPTConfig(
    vocab_size=512,
    block_size=128,
    n_layer=4,
    n_head=8,
    n_kv_head=None,
    d_model=256,
    d_ff=None,
    dropout=0.0,
    tie_weights=True,
    rope_base=10_000.0,
)
```

`n_kv_head` defaults to `n_head`; `d_ff` defaults to `4 * d_model`. Dimensions must be positive, query heads must divide by KV heads, and `d_model` must divide by `n_head`.

`RMSNorm`

```
RMSNorm(dim, eps=1e-5)
```

Computes mean-square variance in FP32, normalizes in the input dtype, and applies a learned weight initialized to ones.

Rotary embeddings

`apply_rope(x, cos, sin)`

Applies rotate-half rotary embeddings to `(batch, heads, sequence, head_dim)`. Cosine and sine tensors broadcast as `(1, 1, sequence, head_dim)`.

`RotaryEmbedding(head_dim, base=10_000.0)`

- Requires an even head dimension.
- Registers inverse frequencies as a non-persistent buffer.
- Caches cosine/sine tensors by sequence length, device, and dtype.
- Recomputes when device, dtype, or required length changes.

`CausalSelfAttention`

```
CausalSelfAttention(config)
```

Uses bias-free Q/K/V/output projections. Key/value width is `n_kv_head * (d_model // n_head)`.

Forward flow:

1. Project to `(B, T, heads, head_dim)` and transpose.
2. Apply RoPE to Q and K.
3. Repeat K/V heads to query-head count with `repeat_interleave`.
4. Use SDPA with `is_causal=True` when no mask is supplied.
5. If an attention mask exists, build a dense additive key-padding plus causal mask.
6. Merge heads, project, and apply residual dropout.

An explicit mask can force less efficient SDPA paths and consume `B × 1 × T × T` memory.

`SwiGLU`

```
SwiGLU(config)
```

Bias-free gate, up, and down projections:

```
down(silu(gate(x)) * up(x))
```

The initial hidden width is `d_ff * 2/3`, rounded upward to a multiple of `n_head`. The comment calls this “head dimension,” but the code uses the number of heads rather than `d_model // n_head`.

`TransformerBlock`

Pre-normalized residual block:

```
x = x + dropout(attention(rms_norm(x)))
x = x + mlp(rms_norm(x))
```

Gradient checkpointing calls `torch.utils.checkpoint` with `use_reentrant=False`, with an older-signature fallback.

`ReferenceTransformer`

Registered as:

```
reference_transformer
gpt
```

Constructor aliases:

```
max_seq_len → block_size
```

Accepted model fields:

```
vocab_size, block_size, n_layer, n_head, n_kv_head,
d_model, d_ff, dropout, tie_weights, rope_base
```

Unknown keywords raise `TypeError`.

Weight initialization

- Linear weights: normal mean 0, standard deviation 0.02.
- Linear biases: zero.
- Embedding weights: normal mean 0, standard deviation 0.02.

When weights are tied, the shared parameter is encountered through both the embedding and LM head during module traversal, so initialization consumes RNG twice.

Gradient checkpointing hook

```
model.set_gradient_checkpointing(True)
```

Propagates the flag to every block.

Forward and loss

```
model(
    input_ids,
    labels=None,
    attention_mask=None,
    **_,
) -> {"logits": tensor, "loss": optional_tensor}
```

- `input_ids` must be `(batch, sequence)`.
- Sequence length cannot exceed `block_size`.
- Labels must be same length or one shorter.
- Same-length labels are shifted to `labels[:, 1:]`.
- Logits are truncated to `[:, :-1]`.
- `-100` labels are ignored.
- Empty/all-ignored labels produce a differentiable zero.

> **⚠ CURRENT CAUSAL ALIGNMENT**
>
> The built-in synthetic/text datasets return one next-token label per input position. Speedtronic 2.0 consumes that already-shifted contract without shifting the labels a second time. Custom models should follow the same convention.

Compatibility aliases

```
GPT = ReferenceTransformer
Transformer = ReferenceTransformer
ReferenceModel = ReferenceTransformer
GPTModel = ReferenceTransformer
TransformerConfig = GPTConfig
```

The top-level package lazily exports only `GPT`, `GPTConfig`, and `ReferenceTransformer`; the other aliases are available from `speedtronic.model`.

GQA shape example

For `d_model=256`, `n_head=8`, and `n_kv_head=2`:

```
head_dim = 32
query projection: 8 × 32 = 256
key/value projection: 2 × 32 = 64
repeat factor: 4
effective K/V heads: 8
```

Custom model guidance

The model registry and trainer do not require this architecture. Follow the custom model tutorial, but remember that registry factories receive the fixed reference-model keyword set and the default config validates reference-oriented dimensions.

Device and precision

`PrecisionPlan`

```
@dataclass(frozen=True)
class PrecisionPlan:
    mode: str
    dtype: Any
    use_scaler: bool
    autocast_enabled: bool
    device_type: str
```

`is_mixed_precision` is true for `bf16` and `fp16`.

`resolve_device(requested="auto")`

Auto order:

1. CUDA when `torch.cuda.is_available()`.
2. MPS when exposed and available.
3. CPU.

Explicit CUDA and MPS requests verify availability and raise `RuntimeError` when unavailable. Other device strings are passed to `torch.device` directly.

`resolve_precision(config, device)`

The resolved plan is used by the trainer; parameters and buffers themselves are not globally cast to the mixed dtype.

Resolution matrix

Device	Request	Resolved mode	Scaler	Autocast
CUDA	`auto`, BF16 supported	`bf16`	No	Yes
CUDA	`auto`, BF16 unsupported	`fp16`	Yes	Yes
CUDA	explicit `bf16`, supported	`bf16`	No	Yes
CUDA	explicit `bf16`, unsupported/error	`fp32`	No	No
CUDA	explicit `fp16`	`fp16`	Yes	Yes
CPU	`auto`	`fp32`	No	No
CPU	explicit `fp32`	`fp32`	No	No
CPU	explicit `fp16`	`fp32`	No	No
CPU	explicit `bf16`	`bf16`	No	Yes
MPS	`auto`	`fp32`	No	No
MPS	explicit mixed mode	`fp32`	No	No
MPS	explicit `fp32`	`fp32`	No	No

> **⚠ EXPLICIT CPU BF16**
>
> The code does not check CPU BF16 capability. It preserves explicit BF16 on CPU and enables CPU autocast. Modern PyTorch may support it, but behavior depends on the installed PyTorch build and custom operators.

`precision.dtype`

`PrecisionConfig.dtype` affects resolution only when `mode == "auto"`. For example:

```
precision:
  mode: auto
  dtype: fp16
```

requests FP16. With `mode: fp32`, `dtype: bf16` is ignored.

`autocast_context(plan, device)`

Returns `nullcontext()` for FP32. For mixed precision, returns `torch.autocast(device_type, dtype)`, with a compatibility fallback for older signatures. The `device` argument is accepted but not directly consulted; the plan's `device_type` controls the context.

`make_grad_scaler(plan)`

Returns a scaler only for FP16 on CUDA. It prefers:

```
torch.amp.GradScaler("cuda")
```

and falls back to `torch.cuda.amp.GradScaler()` for older PyTorch.

The trainer creates a new scaler on every `fit()` invocation, then restores deferred checkpoint state when present.

`supports_fused_adamw(device)`

Returns false on non-CUDA devices. On CUDA it constructs a zero-step fused AdamW optimizer with a one-element parameter and catches `TypeError`, `RuntimeError`, and `ValueError`.

`build_optimizer()` currently builds the actual model on CPU and moves it later. A capability probe on a CUDA parameter can succeed while construction with CPU model parameters fails, causing an ordinary AdamW fallback.

Fused AdamW configuration

```
optimizer:
  fused: null   # probe
```

```
optimizer:
  fused: true   # attempt
```

```
optimizer:
  fused: false  # disable
```

A failed attempt is silent except for the absence of fused behavior.

Performance implications

- BF16 generally avoids CUDA `GradScaler`.
- FP16 CUDA uses dynamic loss scaling.

- CPU auto precision stays FP32 for broad custom-module compatibility.
- MPS auto precision stays FP32.
- Microbatch loss-to-CPU conversion still synchronizes CUDA even when mixed precision reduces tensor compute cost.
- Metric memory queries can also synchronize the current accelerator.

Test evidence

The repository directly tests CPU auto FP32 and explicit CPU FP16 fallback. CUDA, MPS, CPU BF16, fused AdamW, and scaler lifecycle are implementation-derived and not covered by repository tests.

See Local Performance and Runtime and Trainer.

Checkpoints and RNG

`CheckpointError`

A `RuntimeError` subclass used when a loaded artifact is not a state dictionary or no checkpoint exists for copying.

Atomic helpers

`atomic_torch_save(state, path)`

1. Creates parent directories.
2. Writes to `.<name>.tmp-<pid>`.
3. Calls `torch.save`.
4. Replaces the destination with `os.replace`.
5. Removes a leftover temporary file in `finally`.

The rename is atomic on one filesystem, but files and parent directories are not explicitly `fsync`-ed for power-loss durability.

`atomic_json_dump(value, path)`

Writes indented JSON to a PID-specific temporary path and calls `os.replace`. It has no explicit temporary-file cleanup `finally` block.

`CheckpointManager`

```
CheckpointManager(
    directory,
    *,
    every_steps=500,
    keep_last=3,
    enabled=True,
)
```

Requires a positive interval even if disabled.

File layout

```
<directory>/
├── latest.json
├── step_000000000002.pt
├── step_000000000004.pt
└── ...
```

`latest.json`:

```
{"step": 4, "file": "step_000000000004.pt"}
```

`should_save(step)`

True when enabled, step is positive, and `step % every_steps == 0`.

`path_for(step)`

Returns a 12-digit filename: `step_<step:012d>.pt`.

`save(step, state)`

Directly writes the checkpoint, updates `latest.json`, and prunes. It does not itself check `enabled` or `should_save`; the trainer performs those checks.

`latest_path()`

If `latest.json` exists and points to an existing path, it trusts that pointer. Otherwise it scans and sorts `step_<digits>.pt` files and returns the highest step.

A process crash after writing a checkpoint but before updating the pointer can leave the pointer behind a higher valid file.

`load_latest()`

Loads the selected file to CPU with `weights_only=False`, then verifies that the result is a dictionary. It does not try an older checkpoint if the newest is corrupt.

`prune()`

Deletes old `step_*.pt` files to retain `keep_last`. `None` retains all files.

`copy_to(destination)`

Copies the file selected by `latest_path()`. It raises `CheckpointError` when none exists.

Trainer checkpoint schema

```
{
    "model": model.state_dict(),
    "optimizer": optimizer.state_dict(),
    "scaler": scaler_state_or_none,
    "scheduler": scheduler_state_or_none,
    "step": int,
    "samples": int,
    "tokens": int,
    "config": redacted_config_or_none,
    "rng": capture_rng_state(),
    "coordinator": coordinator_state_or_none,
    "precision": {
        "mode": "fp32",
        "autocast_enabled": False,
        "use_scaler": False,
    },
}
```

The stored config and precision plan are informational. Resume does not verify compatibility or reconstruct the current plan from them.

RNG capture

`capture_rng_state()` records:

- Python `random` state;
- NumPy state when available;
- PyTorch CPU RNG state;
- all CUDA RNG states when CUDA is available.

MPS RNG is not captured. DataLoader generators and persistent worker RNG state are also not captured.

Restore behavior

`restore_rng_state()` independently attempts each state restore and ignores individual exceptions.

`Trainer.resume()` loads model, optimizer, scheduler, counters, and RNG immediately. Scaler and coordinator states are deferred:

- scaler state is restored after `fit()` creates the scaler;

- coordinator state is applied after `start()` contacts or reconciles the Hub.

Exact resume granularity

A checkpoint does not include:

- DataLoader iterator position;
- sampler state;
- epoch position;
- persistent worker state;
- arbitrary callback state;
- model train/eval mode.

Finite loaders restart iteration. `shuffle=False` can therefore repeat the beginning of a dataset after resume. RNG restoration does not recreate data-loader position.

Final-step behavior

The trainer only checkpoints exact intervals. A run ending at step 3 with `every_steps=2` has no step-3 checkpoint. Resume returns to the most recent interval checkpoint and can repeat work after it.

Coordinator state

When distributed mode is active, the checkpoint embeds coordinator state. For DumbDiLoCo this includes local/global counters, baseline, unused pending-delta fields, and the complete master outer state when present.

The checkpoint can therefore become large: it may contain the model, optimizer, baseline, global model, and momentum.

v2 pending uploads

DumbDiLoCo checkpoints can include one immutable `pending_upload` job with its step, base outer step, baseline generation, delta, and target snapshot. This makes an in-flight boundary recoverable and bounded to one model-sized CPU copy. Threads, queues, and locks are never serialized. See the DumbDiLoCo coordinator.

Security

`.pt` files are loaded with pickle-capable `torch.load(..., weights_only=False)`. Treat checkpoint directories, distributed state directories, and serialized dataset files as trusted local files. A write-capable local attacker can execute code through a crafted payload.

Operational recommendations

- Use short checkpoint intervals for expensive runs.
- Treat `max_steps` as absolute and retain a stable output directory.
- Copy important checkpoints outside the pruned directory.
- Protect `.pt` files with filesystem permissions.
- Do not use untrusted serialized datasets or state directories.
- Expect approximate rather than exact DataLoader replay.

See Checkpoint and Resume and Security and Limitations.

Metrics, events, and hooks

`TrainingHook`

A structural protocol:

```
class TrainingHook(Protocol):
    def on_event(self, event: str, payload: dict[str, Any]) -> None: ...
```

A plain callable with the same signature is also accepted.

`memory_usage_bytes()`

Attempts to return:

- `torch.cuda.memory_allocated()` when CUDA is available;
- `torch.mps.current_allocated_memory()` when MPS is available;
- otherwise `None`.

The function queries the current/default device rather than accepting the trainer's device explicitly. Memory can therefore be reported for the wrong CUDA device in a multi-GPU setup.

`MetricLogger`

```
MetricLogger(
    level="INFO",
    file=None,
    json_file=None,
    every_steps=1,
    stream=None,
    hooks=None,
)
```

Creates one dedicated Python logger with propagation disabled. It writes to stdout by default or to a text file. JSONL output is optional.

`emit(event, payload)`

1. Copies the payload.
2. Adds invocation-local `elapsed_s` when absent.
3. Adds accelerator `memory_bytes` when absent and available.
4. For `train_step`, returns early when the step is not divisible by `every_steps`.
5. Writes a text log line.
6. Appends one JSON object to `json_file` when configured.
7. Dispatches to every hook.
8. Catches and logs hook exceptions.

Cadence filtering occurs before hook delivery, so hooks do not receive skipped training steps.

Memory is queried before cadence filtering, so skipped steps can still incur a device query.

Text output

```
2026-01-01 12:00:00 | INFO | train_step step=1 loss=1.2 lr=0.0003 ...
```

JSONL output

Each line is:

```
{"event":"train_step","step":1,"loss":1.2,"elapsed_s":0.01}
```

Non-JSON-native values use `default=str`.

`close()`

Flushes and removes the logger's Speedtronic handler and closes file handlers. `train_from_config()` does not call `close()` automatically, so repeated programmatic runs can retain open resources until garbage collection.

`CallbackList`

```
CallbackList(callbacks=None)
```

Fan-out adapter whose `on_event` invokes each callback's `on_event` method or callable form.

Event catalog

Event	Producer	Typical payload
`train_start`	Trainer	start/target step, device, precision, accumulation, parameter count
`train_step`	Trainer	loss, LR, rates, counters, microbatches
`checkpoint`	Trainer	step, checkpoint path
`train_end`	Trainer	absolute steps, elapsed time, final loss
`compile`	Trainer	enabled flag and failure reason
`outer_step`	Master outer loop	outer step, candidates found, deltas included
`delta_upload_queued`	Coordinator	local step, base outer step
`delta_uploaded`	Async uploader	local step
`delta_upload_skipped`	Coordinator	step and overflow/in-flight reason
`delta_upload_failed`	Async uploader	step and error
`delta_upload_stale`	Async uploader	step and baseline generation
`shape_profile`	Runtime	device, precision, alignment, warning count
`shape_warning`	Runtime	field, value, suggested alignment
`ooo_backprop`	Trainer	enabled, reason, streams, stage names

Hook adapter

`speedtronic.hooks.on_event(callback)` returns a two-argument wrapper. It is useful when adapting a function whose natural signature differs from `(event, payload)`.

W&B adapter

```
from speedtronic.integrations import WandbHook

hook = WandbHook(project="my-project")
```

Requires the `logging` extra or `wandb`. Construction calls `wandb.init`; every event calls `wandb.log({**payload, "event": event})`.

The adapter has no `close()` method and does not finish the run.

TensorBoard adapter

```
from speedtronic.integrations import TensorboardHook

hook = TensorboardHook(log_dir="runs/tensorboard")
```

Requires `tensorboard` or PyTorch's TensorBoard writer. It writes each numeric payload value to:

```
<event>/<key>
```

Call `hook.close()` explicitly. `MetricLogger.close()` does not automatically call hook close methods.

Attaching hooks through configuration

YAML does not have a logging hook field. The runtime's `build_logger()` does not attach W&B or TensorBoard.

Programmatic pattern:

```
from speedtronic.integrations import WandbHook
from speedtronic.profiling import MetricLogger
from speedtronic.runtime import build_runtime

trainer, _ = build_runtime(config)
hook = WandbHook(project="speedtronic")
trainer.logger.hooks.append(hook)
result = trainer.fit()
hook._run.finish()  # adapter currently has no public close
```

Direct construction is also possible:

```
logger = MetricLogger(hooks=[WandbHook(project="speedtronic")])
trainer = Trainer(..., logger=logger)
```

Hook failure semantics

Hook exceptions are isolated from training. The logger writes a warning and continues. This protects the loop but can hide broken observability integrations unless warnings are monitored.

Concurrency caveat

DumbDiLoCo's master outer thread can emit `outer_step` while the training thread emits other events. `MetricLogger` does not synchronize JSONL appends or hook calls across those threads.

See Observability Tutorial and DumbDiLoCo.

Extension points

Model registry

`ModelRegistry`

```
registry = ModelRegistry()
registry.register(name, factory=None)
registry.get(name)
registry.names()
registry.build(name, **kwargs)
```

`register()` works as a decorator or direct function. Registration silently overwrites an existing name.

Built-in lazy loading

The package-level global registry knows that `reference_transformer` and `gpt` are built-ins. If a name is absent, `get()` imports `speedtronic.model`; the model decorators then populate the **global** registry.

A newly constructed `ModelRegistry()` advertises built-in names through `names()` but importing the model does not register factories into that separate instance. Use the exported global `registry` for normal custom registration.

`register_model(name, factory=None)`

Public decorator/function backed by the global registry.

```
import torch
from speedtronic import register_model

@register_model("my_model")
def make_model(**kwargs):
    return torch.nn.Linear(kwargs["d_model"], kwargs["vocab_size"])
```

`build_model(config, **overrides)`

Always forwards these fields:

```
vocab_size
block_size
max_seq_len-derived model context
n_layer
n_head
n_kv_head
d_model
d_ff
dropout
tie_weights
rope_base
```

Explicit `overrides` replace standard fields. Custom factories should accept `**kwargs` or deliberately ignore unrelated reference-model parameters.

Configuration-driven custom models

Unknown custom model-specific configuration keys are rejected. A model factory must obtain architecture-specific values inside Python or through the standard fields.

Registration is process-local. A CLI process loading only YAML has no plugin import mechanism and cannot discover a factory registered only in another process.

To make a CLI-visible model:

1. Put registration in an importable Python module.
2. Import that module before `build_runtime()`.
3. Construct the runtime programmatically, or add a supported plugin loader in the application.

Trainer model contract

Accept a dictionary batch as keyword arguments or a tuple batch as `(inputs, labels)`. Return a direct loss, mapping with loss, or compatible causal logits. See Runtime and Trainer.

Gradient checkpointing convention

Model hook

```
def set_gradient_checkpointing(self, enabled: bool = True):
    ...
```

The trainer calls the hook once during construction. Missing or failing hooks warn and continue.

Public helper

```
from speedtronic import set_gradient_checkpointing

set_gradient_checkpointing(model, True)
```

`module_utils.set_gradient_checkpointing()` raises `AttributeError` when the model does not expose the hook, unlike the trainer's warning-only behavior.

Dataset extension

Use:

```
build_dataloader(config.data, dataset=my_dataset, tokenizer=my_tokenizer)
```

The built-in collator recognizes causal dictionaries, equal-width tuples/lists, and tensors. For other tasks, build a `DataLoader` directly and pass it to `Trainer`.

Sync coordinator extension

Implement:

```
class MyCoordinator:
    def start(self): ...
    def after_optimizer_step(self, model, step): ...
    def stop(self): ...
    def state_dict(self): ...
    def load_state_dict(self, state): ...
```

Contract details:

- `start()` is called only when work remains.
- `after_optimizer_step()` runs after the local scheduler step.
- A true result can clear optimizer state.
- `stop()` runs in `finally`.
- `state_dict()` is embedded in the next local checkpoint.
- `load_state_dict()` runs after coordinator startup.

Metric hooks

A hook can be:

```
def hook(event: str, payload: dict[str, Any]) -> None:
    ...
```

or an object with `on_event`. Hook failures do not stop training.

See Observability for built-in W&B and TensorBoard adapters.

Optional integrations

Adapter	Dependency	Construction	Close behavior
`WandbHook`	`wandb`	Calls `wandb.init`	No public close method
`TensorboardHook`	TensorBoard writer	Creates `SummaryWriter`	Has `close()`

Install with:

```
pip install 'speedtronic[logging]'
```

Stability categories

Category	Examples
Public top-level API	`SpeedtronicConfig`, `Trainer`, `register_model`, `HubClient`
Public module API	`build_dataloader`, `CheckpointManager`, tensor helpers
Compatibility alias	`GPT`, `TrainingEngine`, `DumbDiLoCo`, `HubTransport`, `Config`
Declarative but currently unused	`SyncResult`, `OuterState`, pending coordinator delta fields
Internal extension seam	`SyncCoordinator` protocol and model checkpoint hook
Private implementation	Names beginning with `_`; not stable API

The generated inventory lists all of these declarations and source lines.

Speedtronic v2

Version 2.0.0 adds optimization and systems techniques that can be enabled independently while preserving CPU, CUDA, and MPS fallbacks.

What is new

Area	Feature	Default
Optimizer	Hybrid Muon + AdamW routing	AdamW
Optimizer	Muon+ post-orthogonal normalization	Off
Optimizer	Cautious sign-aligned update wrapper	Off
Systems	CUDA stream-backed out-of-order backprop (conservative)	Off
Systems	Startup shape-alignment warnings	On when applicable
Distributed	Asynchronous delta dispatch and global polling	On

Design rules

- Muon's Newton–Schulz iteration uses only PyTorch operations.
- CPU and MPS never require CUDA streams; they use the standard path.
- Shape findings are warnings, not startup failures.
- Distributed Hub work is dispatched away from the training thread by default.
- Features compose, but `torch.compile` and stream scheduling are mutually exclusive in this release.
- Deferred techniques are documented explicitly rather than exposed as dead configuration flags.

Pages

- Deferred and roadmap decisions

v2 optimizers

Configuration

AdamW remains the default. Use the mapping form for full control:

```yaml
optimizer:
  name: muon
  lr: 0.0003
  weight_decay: 0.1
  muon_plus: true
  cautious: true
```

The addendum's scalar form is also accepted:

```yaml
optimizer: muon
muon_plus: true
cautious: true
```

`muon_plus` is valid only with `name: muon`. `cautious` composes with either base optimizer.

Hybrid routing

Muon is intended for hidden 2-D matrix weights:

- attention Q/K/V/output projections;
- MLP gate/up/down projections;
- neutral 2-D custom matrix parameters.

These stay on AdamW:

- embeddings and positional embeddings;
- LM heads, classifiers, and output projections;
- RMSNorm/LayerNorm/BatchNorm parameters;
- biases, scalars, and non-2D tensors;
- frozen parameters.

The bundled reference transformer routes seven hidden matrices to Muon and four unique parameters to AdamW when weights are tied.

A module can explicitly override the heuristic:

```
module._speedtronic_optimizer_role = "muon"  # or "adamw"
```

The role must be valid for a floating 2-D parameter.

Newton–Schulz

`newton_schulz(matrix, steps=5, eps=1e-7)` uses the quintic iteration with coefficients `(3.4445, -4.7750, 2.0315)` in pure PyTorch. Tall matrices are transposed internally, normalized, iterated, and transposed back. Zero matrices remain finite.

There is no SVD, Triton kernel, custom CUDA extension, or hardware-specific dependency.

Muon update

For each routed matrix:

```
B ← momentum · B + (1 - momentum) · gradient
Q ← lerp(gradient, B, momentum)       # Nesterov form
U ← NewtonSchulz(Q)
W ← (1 - learning_rate · weight_decay) · W
    - learning_rate · max(1, rows / columns)^0.5 · U
```

Momentum state is stored as FP32 (or higher) independently of parameter precision. Sparse and complex gradients are rejected with a clear error.

Muon+

Muon+ adds post-orthogonalization normalization. The default `row_col` mode first normalizes rows, then columns, with an epsilon inside each square root. It adds no persistent optimizer state.

```
optimizer:
  name: muon
  muon_plus: true
```

Muon may change which solution a model selects, not only how quickly it trains. Its convergence/simplicity-bias trade-off is an active research question, so it is not a guaranteed free lunch.

Cautious updates

With `cautious: true`, Speedtronic snapshots parameters, lets the base optimizer compute its update, and keeps entries whose update direction aligns with the current gradient:

```
mask = (observed_update · gradient > 0)
parameter = old - masked_update
```

The wrapper adds no mask history to the checkpoint. Because the wrapper is composable with arbitrary optimizers, the observed update includes the base optimizer's decoupled weight-decay contribution; native Muon and AdamW state remain intact.

Scheduler and checkpoint behavior

The hybrid optimizer exposes both groups through one outer optimizer, so the existing `LambdaLR` updates both learning rates. Its state dictionary contains ordinary Adam moments and Muon momentum buffers. Clearing inner optimizer state after a distributed sync clears both branches.

Cautious is a facade over the base optimizer and delegates its state, zero-grad, and checkpoint operations.

CPU/CUDA/MPS behavior

Muon and Muon+ are pure PyTorch and work on all three devices. Fused AdamW is only attempted for CUDA and can silently fall back to ordinary AdamW. Cautious wrapping disables the fused path so the observed update is available for masking.

Evidence

The v2 test suite covers routing, tied weights, zero/rectangular Newton–Schulz matrices, Muon+ normalization, hybrid stepping, cautious wrapping, sparse rejection, scheduler updates, and CPU runtime smoke training. CUDA fused and stream paths remain hardware-dependent and are documented separately.

Out-of-order backprop

```
ooo_backprop: true
ooo_streams: 4
```

The feature is off by default and opt-in because its benefit is model and hardware dependent.

What PyTorch already does

PyTorch's autograd engine already computes ready nodes from the autograd DAG rather than forcing a strict Python reverse-layer loop. Its remaining stream behavior follows the stream that created each forward node. Speedtronic's v2 scheduler supplies streams to disjoint module stages, records producer dependencies, and synchronizes each stage back to the ambient stream before an untracked parent operation runs. This preserves correctness while keeping the scheduling boundary explicit. It is a conservative first implementation and does not promise kernel overlap on every model.

Stage selection

1. A model may expose `ooo_stage_names`, a sequence of dotted module names.
2. Otherwise the scheduler expands container modules into disjoint leaf stages.
3. Each stage is assigned round-robin to at most `ooo_streams` streams.

For every stage:

- inputs wait on the recorded producer stream;
- tensors crossing streams use `record_stream` for allocator safety;
- stage outputs are associated with their producer stream;
- the ambient stream waits for each stage before untracked parent/container operations continue, avoiding a read of an unfinished stage;
- the current stream waits for all stage streams before host-visible loss use.

Failure and fallback

Condition	Behavior
CPU or MPS	Standard sequential backward; event reports disabled
CUDA unavailable	Standard sequential backward
Fewer than two stages	Standard sequential backward
Setup failure	Hooks removed, warning/event emitted, training continues
`compile: true`	OOO disabled with a warning; compiled path wins
CUDA graph capture	Not supported by this scheduler

Honest performance expectations

Published out-of-order backprop work reports roughly 1.03–1.58× single-GPU throughput depending on the model, but those results use a lower-level implementation. Speedtronic's first pure-PyTorch path prioritizes dependency correctness and conservative synchronization; it is intended to provide a portable scheduling seam, not a fixed multiplier. A compute-saturated model may see no speedup, and a small model may see only kernel-overhead changes.

Speedtronic does not claim a fixed multiplier. Benchmark warm-up separately from steady state and compare against the same model, shapes, precision, and seed.

Reference-model behavior

The bundled transformer is mostly a sequential chain at block granularity. Its value is still explicit stream placement and reduced ambiguity about producer dependencies, but it cannot expose a large reorderable DAG. Custom models with independent towers or branches have more scheduling freedom.

Correctness

The feature changes stream placement, not gradient mathematics. Autograd's `.grad` accumulation, gradient scaling, clipping, accumulation, and coordinator boundaries remain in the trainer. CPU parity is the default test path; CUDA parity and race testing should be run on the target accelerator before enabling the flag in production.

Interaction with compilation

`ooo_backprop` and `torch.compile` are mutually exclusive in v2. Dynamo does not expose a stable API for tracing arbitrary `torch.cuda.stream` contexts inside a compiled region. Choose one scheduling mode and benchmark it.

AoT scheduling decision

The addendum's ahead-of-time kernel scheduling item is **deferred as subsumed** for this release. Inductor already performs graph capture, fusion, topological scheduling, and autotuning inside `torch.compile`. Speedtronic does not expose a duplicate `aot_scheduling` flag that would depend on private PyTorch APIs or claim a hardware-specific speedup.

Shape validation

Shape validation catches awkward tensor-core/GEMM dimensions before they silently reduce throughput.

```yaml
shape_validation:
  enabled: true
  alignment: auto
  check_batch: true
  check_sequence: true
  check_model: true
  check_vocab: false
  warn_on_cpu: false
```

Automatic profiles

Device and resolved precision	Automatic alignment
CUDA FP16/BF16	8
CUDA FP32/TF32	None by default; set an explicit alignment for a benchmark profile
CPU	None by default
MPS	None by default

An explicit positive `alignment` overrides the profile. On CPU or MPS, set `warn_on_cpu: true` to enable that explicit benchmark profile. `"none"` disables warnings.

Inspected values

- `data.micro_batch_size`;
- `data.block_size`;
- `micro_batch_size * block_size`;
- `nn.Linear` input/output features;
- embedding dimensions;
- 2-D convolution channels;
- optionally vocabulary/output width.

Warnings are deduplicated and include a nearby upward-aligned suggestion when that suggestion remains valid for the configured target batch. They never change configuration, memory use, or model architecture. Set `alignment: none` to disable them.

Example

```
shape warning: data.micro_batch_size=3 is not a multiple of 8; consider 8 (kernel batches near this
alignment are usually more efficient)
```

If the nearby micro-batch suggestion would violate the target-batch divisibility rule, `suggested` is emitted as `null` instead.

The warning is a hint, not an error. A user may intentionally choose an unaligned shape for memory, debugging, or data semantics.

Custom models

A model may expose:

```python
def speedtronic_shape_metadata(self) -> Mapping[str, int]:
    return {"block1.mlp_hidden": 300}
```

The validator falls back to standard PyTorch module introspection when the hook is absent or fails. Shape metadata is observational and has no effect on training.

Runtime integration

`build_runtime()` resolves precision first, runs validation, then constructs the optimizer. Direct `Trainer` users receive the same validation on the first `fit()` call. The `train_start` event includes alignment and warning count.

Migrating to 2.0.0

Version and package identity

```
speedtronic==2.0.0
```

The PyPI distribution and import package remain exactly `speedtronic`.

Optimizer migration

AdamW remains the default. No v1 configuration needs to change.

```yaml
optimizer: muon
muon_plus: true
cautious: true
```

For explicit fields:

```yaml
optimizer:
  name: muon
  muon_plus: true
  cautious: true
  muon_momentum: 0.95
  muon_ns_steps: 5
  muon_norm_eps: 1.0e-8
```

Muon receives hidden 2-D matrices; embeddings, heads, normalization weights, and biases remain on AdamW. Checkpoint state therefore contains both AdamW moments and Muon momentum when Muon is enabled.

Causal labels

Speedtronic 2.0 consumes the already-shifted labels emitted by its datasets. The reference model no longer shifts same-length labels a second time. Custom models should return one next-token label per input position, or use the model's own loss mapping.

Scheduler target

A run target shorter than the historical 1000-step scheduler default now synchronizes the default schedule to `run.max_steps`. Set `scheduler.max_steps` explicitly when a longer schedule is intentional.

Distributed defaults

DumbDiLoCo now defaults to:

```yaml
distributed:
  async_delta_upload: true
  async_global_poll: true
  delta_upload_queue_size: 1
  delta_upload_overflow: skip
  delta_upload_shutdown_timeout: 5.0
```

Set both async flags to `false` for the v1 synchronous behavior during debugging or deterministic transport tests.

Shape and OOO settings

```yaml
shape_validation:
  enabled: true

ooo_backprop: false
ooo_streams: 4
```

OOO backprop is opt-in and unavailable as an active path on CPU/MPS; those devices use standard sequential backward.

Checkpoint compatibility

Old checkpoints remain readable where their model and optimizer state shapes match. Muon/Muon+/cautious state is not interchangeable with an AdamW-only checkpoint. Resume with a changed optimizer algorithm should start a new run or use a deliberately migrated checkpoint.

The bundled trainer checkpoint now redacts configuration secrets and includes DumbDiLoCo pending-upload state when a job is in flight.

Deferred items

Sophia, mixture-of-experts layers, FP8 training, and multi-GPU pipeline or tensor parallelism are not part of 2.0.0. AoT scheduling is documented as subsumed/deferred by `torch.compile` rather than exposed as a duplicate flag.

Deferred techniques and roadmap

The v2 addendum deliberately separates implemented work from ideas that require more evidence. Speedtronic records those decisions so they do not become undocumented dead flags.

Implemented in 2.0.0

- Hybrid Muon and AdamW routing.
- Muon+ post-orthogonalization normalization.
- Cautious update masking.
- CUDA stream-backed dependency-aware backprop scheduling.
- Startup shape-alignment warnings.
- Non-blocking DumbDiLoCo delta dispatch and global polling.

AoT kernel scheduling: deferred/subsumed

`torch.compile` and Inductor already perform graph capture, fusion, topological scheduling, and autotuning. A second public AoT flag would duplicate that work and would require private PyTorch APIs or CUDA-specific dependencies. Speedtronic records the item as **deferred/subsumed** rather than shipping a flag that could not be portable or honestly benchmarked.

Sophia: deferred

Sophia's second-order Hessian estimate can help some model scales, but its complexity and adoption consistency do not yet justify adding a second optimizer family to the v2 release. It remains a candidate for a future version after Muon integration is validated.

MoE layers: deferred

MoE is an architecture decision for a reference-model variant, not a training engine technique. It would affect parameter routing, memory, and checkpoint compatibility.

FP8: deferred and hardware-detected if revisited

FP8 requires newer accelerator classes and must not become a default or a hardware-locked dependency. Any future support would be capability-detected and opt-in, following the existing BF16 detection pattern.

Multi-GPU parallelism: unchanged non-goal

Pipeline and tensor parallelism remain out of scope, as do FSDP-style partitioning and a hosted dashboard.

Muon convergence caveat

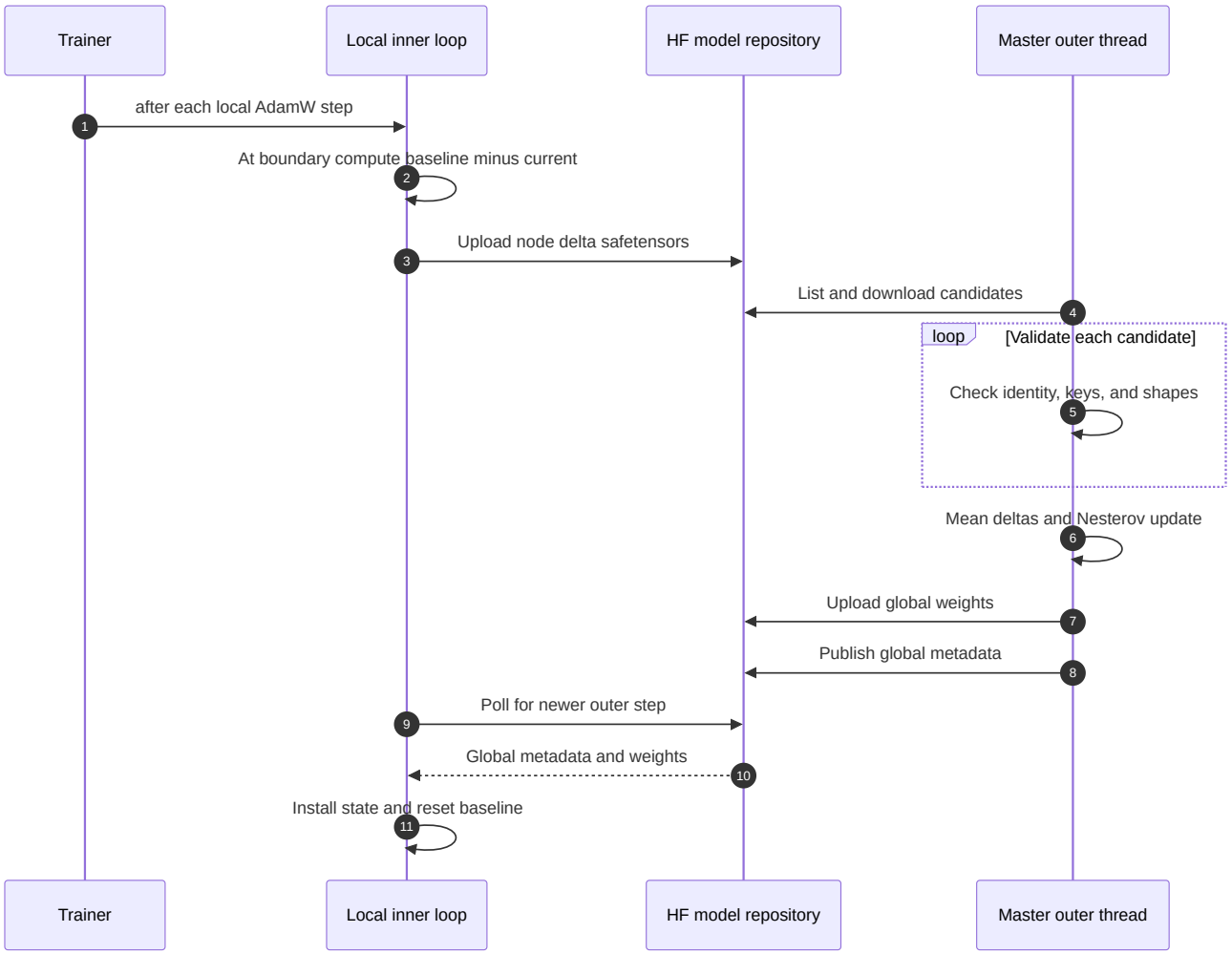
Muon's reported speed advantage may come with a different simplicity bias or selected solution. Users should compare quality and not only steps per second.

DumbDiLoCo overview

DumbDiLoCo is Speedtronic's optional asynchronous distributed mode. Nodes train locally, upload pseudo-gradient deltas to a Hugging Face model repository, and periodically install newer global weights. A master aggregates deltas and publishes a new global model.

It does not use `torch.distributed`, NCCL, collectives, a parameter server, a shared clock, or straggler barriers.

Topology



Roles

Role	Local training	Uploads deltas	Runs outer aggregation	Owns global state
`master`	Yes	Yes	Yes, background daemon thread	Yes
`worker`	Yes	Yes	No	No
`single`	N/A when disabled	No	No	No

A master is also a local worker.

Terminology

Term	Meaning
Local step	One completed local AdamW update after accumulation
Inner step	Another name for a local optimizer update
Inner boundary	`local_step % inner_steps == 0`
Baseline	State at the last successful upload or global installation
Pseudo-gradient	`float32(baseline) - float32(current)`
Outer round	One successful master aggregation/publication
Outer step	Integer version in `global/step_count.json`
Global model	Complete cloned `state_dict`
Node delta	Floating/complex state deltas plus safetensors metadata

Minimal master config

```yaml
distributed:
  enabled: true
  mode: dumb_diloco
  role: master
  node_id: master-1
  collaborators: []
  repo_id: your-org/your-run
  token: null
  inner_steps: 500
  poll_interval: 60
  outer_lr: 0.7
  outer_momentum: 0.9
  state_dir: .speedtronic/diloco
  reset_inner_optimizer: true
  async_delta_upload: true
  async_global_poll: true
```

Prefer `HF_TOKEN` or another Hugging Face SDK credential mechanism over YAML tokens.

Minimal worker config

```yaml
distributed:
  enabled: true
  mode: dumb_diloco
  role: worker
  node_id: worker-1
  repo_id: your-org/your-run
  token: null
  inner_steps: 500
  poll_interval: 60
  reset_inner_optimizer: true
```

Every participant needs a unique, explicit, path-safe node ID.

Inner-loop lifecycle

At every optimizer step:

1. Trainer advances the local scheduler.
2. Coordinator receives the new local step.
3. On a boundary, compute a cumulative pseudo-gradient.
4. Upload the delta.
5. Poll for a newer global version.
6. Install compatible global state if available.
7. Return whether the trainer should reset optimizer state.

Between boundaries, polling occurs no more than once per `poll_interval`.

Outer-loop lifecycle

The master outer thread:

1. Lists remote delta paths.
2. Downloads candidates.
3. Skips processed identities.
4. Validates keys and shapes.
5. Averages all valid deltas with equal weight.
6. Applies Nesterov momentum SGD.
7. Publishes full global weights.
8. Publishes outer-step metadata.
9. Saves local master state.
10. Notifies the training thread through a snapshot callback.

The training thread installs snapshots at normal post-step boundaries; the outer thread never mutates the model during a forward/backward operation.

Hub repository layout

```
<repo_id>/
├── global/
│   ├── latest.safetensors
│   └── step_count.json
└── nodes/
    ├── <node-a>/
    │   ├── delta_500.safetensors
    │   └── delta_1000.safetensors
    └── <node-b>/
        └── delta_500.safetensors
```

See Hub Transport and Tensor Format for exact fields.

Asynchrony boundary

Only the master's outer aggregation runs in a background thread. Worker startup, polls, and uploads execute synchronously on the training thread. Hub retries can therefore delay a local loop even though nodes do not wait for stragglers or fixed rounds.

v2 non-blocking transport

In 2.0, delta dispatch is asynchronous by default. The training thread snapshots state, computes the delta, and enqueues one immutable job; a daemon uploader performs Hub I/O. A second boundary skips and logs while a job is in flight. Global polling uses a separate background lane and installs downloaded state only on the training thread.

Set `async_delta_upload: false` and `async_global_poll: false` for the v1 synchronous behavior. See Coordinator and migration.

- All valid deltas in one poll receive equal weight.
- There is no staleness weighting, token weighting, participant weighting, or quorum.
- Deltas can be based on stale or unrelated global versions.
- Nodes do not coordinate a common membership snapshot.
- Non-floating buffers are published globally but excluded from deltas and outer updates.

Checkpoint interaction

A local trainer checkpoint can include the coordinator baseline and the full master outer state. Worker restart behavior begins from the latest readable global model; exact mid-inner-loop recovery is not implemented.

Security model

> **TRUSTED WRITERS ONLY**
>
> Any account with repository write access can replace global weights, deltas, metadata, or referenced files. There is no cryptographic node identity, signature, checksum binding, Byzantine defense, outlier rejection, or public-participation model.

- Create and verify a private repository.
- Use a dedicated service account.
- Grant write access only to trusted workers.
- Use explicit unique node IDs.
- Protect local state/cache directories.
- Never place tokens in version-controlled YAML.
- Operate one active master for a repository.

Current limitations

- Fixed global weight and metadata paths are separate uploads, not an atomic transaction.
- Historical deltas are never deleted.

- The master redownloads all listed candidates to inspect metadata before deduping.
- The processed-delta ledger is local, not published in Hub metadata.
- Multiple masters can overwrite each other.
- Nodes with the same seed can see identical data unless users provide different data/seed.
- Complex tensors are treated through float32 conversion rather than true complex arithmetic.
- A stopped coordinator can be started again by a later trainer call; bounded shutdown may leave a daemon Hub request running until it returns.

Continue with Coordinator, Hub Transport, Outer Loop, and Tensor Format.

DumbDiLoCo coordinator

Public API

```
from speedtronic import DumbDiLoCoCoordinator
# alias: DumbDiLoCo
```

Constructor:

```
DumbDiLoCoCoordinator(
    config,
    model,
    *,
    hub=None,
    state_dir=None,
    logger=None,
)
```

`config` may be:

- `DistributedConfig`;
- a full `SpeedtronicConfig` exposing `.distributed`;
- a root mapping containing a `distributed` section.

The package-level `SyncResult` dataclass exists with `pushed`, `loaded_global`, and `outer_step`, but the current coordinator does not instantiate or return it.

Node identity and paths

Identity resolution:

```
config.node_id
→ HF_USERNAME
→ USER
→ "local"
```

Explicit IDs are config-validated. Environment-derived IDs are not revalidated.

State root:

```
<state_root>/<node_id>/
```

`build_runtime()` resolves a relative `state_dir` under `run.output_dir`. The Hub `cache_dir` remains relative to the process working directory.

Constructor state

The constructor captures an initial CPU baseline, initializes counters, and may create a `HubClient` or use an injected fake/transport.

Properties:

```
coordinator.local_step
coordinator.last_global_step
coordinator.outer_step
```

`start()`

The first call starts the role-specific lifecycle. `stop()` releases the started flag after bounded shutdown, so a later `fit()` can start fresh background lanes. If a Hub request outlives the timeout, its daemon thread continues until the request returns and emits a shutdown warning.

Master startup

1. Attempt private repository creation with `exist_ok=True`.
2. Attempt `write` collaborator grants.
3. Construct `MasterOuterLoop`; this loads local master state.
4. Read remote global metadata.
5. Select local state, remote state, or bootstrap.
6. Start the background outer thread.

Repository and collaborator failures do not stop local training.

Worker startup

1. Read global metadata.
2. Download/install the referenced global if available.
3. Establish a new baseline.
4. Continue locally when metadata is absent or the read fails.

Startup state selection

Let `L` be local outer step and `R` remote outer step.

Condition	Selection
`L > 0` and remote unavailable or `R <= L`	Local global state wins
`R > L`	Remote model wins; outer state adopts it and resets momentum
`L = 0`, `R = 0`	Remote model is installed into the local model
No local state and no remote metadata	Bootstrap current model at outer step 0

Remote state does not contain processed deltas or Nesterov momentum.

Global installation

`_install_state()` filters incoming tensors to keys present locally with identical shapes, then calls:

```
model.load_state_dict(filtered, strict=False)
```

Missing keys, extra local keys, and shape mismatches are retained/ignored rather than raising. After installation, the complete local model becomes the new baseline.

Polling

`after_optimizer_step(model, step)` verifies model identity, records local step, and computes:

```
boundary = step % inner_steps == 0
```

At a boundary it uploads and force-polls. At other steps it polls only if `poll_interval` elapsed since the last poll.

Worker poll:

1. Read metadata.
2. Ignore versions `<= last_global_step`.
3. Use a cached per-version safetensors file when valid.
4. Delete and redownload a corrupt cache entry.
5. Install state and advance the baseline.

Master poll:

1. Read the outer loop's local snapshot under its lock.
2. Install it when newer.

3. Otherwise use the callback snapshot fallback.

Delta upload

At an inner boundary:

```
current = cpu_state_dict(model.state_dict())
delta = compute_pseudo_gradient(baseline, current)
hub.upload_delta(... base_outer_step=last_global_step ...)
```

Only after successful upload does the coordinator:

- set `_last_uploaded_step`;
- advance the baseline to `current`.

A failed upload leaves the baseline unchanged. The next boundary can therefore recompute a cumulative delta unless a global installation resets the baseline first.

v2 asynchronous dispatch

With `async_delta_upload: true` (the v2 default), a boundary copies the model state, computes the pseudo-gradient, and calls `put_nowait` on a one-slot queue. The uploader thread performs serialization and Hub retries. The next optimizer step is not held behind that network call. A second boundary while the slot is occupied returns without waiting and emits `delta_upload_skipped`.

A successful completion advances the baseline to the immutable target snapshot only when the baseline generation is unchanged. A failed upload leaves the baseline in place, so the next accepted boundary can send a cumulative delta. A global installation increments the generation and prevents an older in-flight completion from overwriting the new baseline.

`async_global_poll: true` submits metadata/state fetches to a separate worker. The uploader/poll workers never mutate the live model; global state is installed at the next training-thread callback.

Optimizer-reset signal

The method returns:

```
pushed or loaded
```

The trainer variable is named `weights_replaced`, but a true return can mean either:

- a delta upload succeeded; or

- newer global state was installed.

When `reset_inner_optimizer=true`, the trainer clears `optimizer.state` after any true result. This can occur mid-inner-loop after a global poll.

State serialization

`state_dict()` returns:

```
node_id
role
last_global_step
last_uploaded_step
local_step
baseline
pending_delta
pending_delta_step
outer (master only)
```

`pending_delta` and `pending_delta_step` are currently never populated by normal upload logic. `last_uploaded_step` is informational and does not suppress repeated uploads.

`load_state_dict()` restores counters, baseline, pending fields, and optional master outer state. It does not enforce that saved role/node ID matches the current process.

Resume ordering caveat

The trainer:

1. calls `coordinator.start()`, which may install and baseline a fresh global;
2. then calls `coordinator.load_state_dict()` with deferred checkpoint state.

The checkpointed baseline can overwrite the baseline established by fresh global installation. On a worker resume, that can pair a newer model with an older baseline and produce a large unintended pseudo-gradient. This is a correctness limitation addressed by v2's prepared startup state.

Stop behavior

`stop()` marks the coordinator stopped and asks the master outer thread to stop. It:

- does not force a partial delta;
- does not force a final outer sync;
- joins the thread for up to 10 seconds;
- does not clear `_started`.

A worker retains its latest local/global state. A master retains its local outer state after successful publication/save.

Failure semantics

Failure	Behavior
Initial repository/create/grant	Warning; local loop continues
Initial worker Hub read	Warning; local weights remain
Poll failure	Warning; model unchanged
Key/shape mismatch in delta	Error log; no upload
Upload failure	Warning; baseline not advanced
Corrupt local global cache	Delete and redownload
Wrong model passed to callback	`ValueError` stops training

Direct construction

```
from speedtronic.config import DistributedConfig
from speedtronic.distributed import DumbDiLoCoCoordinator

config = DistributedConfig(
    enabled=True,
    role="worker",
    node_id="worker-1",
    repo_id="org/run",
)
coordinator = DumbDiLoCoCoordinator(config, model)
coordinator.start()
```

A real Hub token is resolved by the Hugging Face SDK; Speedtronic itself passes the configured token and does not define a separate token environment lookup.

Related: Overview, Outer Loop, and Security.

Hub transport

Public API

```
from speedtronic import HubClient
# distributed alias: HubTransport
```

Errors:

```
HubError(RuntimeError)
HubUnavailable(HubError)
```

Constructor

```
HubClient(
    repo_id,
    *,
    token=None,
    cache_dir=None,
    api=None,
    retry_initial=1.0,
    retry_max=60.0,
    retry_attempts=6,
    sleeper=time.sleep,
)
```

`repo_id` is required. The client always treats it as a Hugging Face **model** repository. `api` and `sleeper` are injection points for tests.

The `api` property lazily constructs `huggingface_hub.HfApi(token=token)`. If the SDK is missing, operations raise `HubUnavailable`.

Retry behavior

Every network/repository operation is wrapped by `_retry`:

```
attempt immediately
→ wait retry_initial
→ double delay after each failure
→ cap at retry_max
→ raise HubUnavailable after retry_attempts
```

With defaults, sleeps are 1, 2, 4, 8, and 16 seconds before the sixth final attempt.

The wrapper catches every `Exception`, including permanent authentication, malformed request, and programming errors. There is no jitter, retry classification, or explicit request timeout.

Repository methods

`create_repo(private=True)`

Calls:

```
api.create_repo(
    repo_id,
    repo_type="model",
    private=private,
    exist_ok=True,
)
```

`exist_ok=True` does not verify that an existing repository is private.

`add_collaborator(username, permission="write")`

Valid permissions:

```
read
write
admin
```

Attempts the newer keyword API, then a positional compatibility form. Missing SDK support raises `HubError`.

`list_files()`

Returns sorted repository filenames.

`file_exists(remote_path)`

Builds a set from `list_files()` and checks membership. This is a full repository listing, not a direct existence API.

`upload(local_path, remote_path)`

Verifies local existence and calls `api.upload_file` with compatibility handling.

`download(remote_path, local_path=None)`

1. Creates a temporary directory under `cache_dir`.

2. Uses an API download method when available, then the top-level SDK helper.
3. Copies the downloaded file to the destination.
4. Removes the temporary directory in `finally`.

The final copy is not atomic.

JSON methods

`read_json(remote_path, default=None)`

Checks `file_exists`, downloads to `cache_dir/json/<remote_path>`, then parses JSON.

`write_json(remote_path, value)`

1. Writes pretty, sorted JSON under `cache_dir/outgoing`.
2. Uses one deterministic `.tmp` path.
3. Replaces the local staging file.
4. Uploads it.
5. Copies the local result into the read cache.

The deterministic temporary filename is not process-safe.

Global metadata

```python
@dataclass(frozen=True)
class GlobalMetadata:
    outer_step: int
    updated_at: str
    model_file: str = "global/latest.safetensors"
```

`from_dict()` accepts legacy `step`, missing metadata defaults, and coerces values to int/str. `updated_at` is informational; synchronization decisions use `outer_step`.

`global_metadata()`

Attempts to read `global/step_count.json`. On `HubUnavailable`, it can fall back to cached JSON. It can also return cached metadata when the remote path is absent.

Malformed cached JSON is not converted into `HubUnavailable`.

Global publication

```
publish_global(
    state,
    *,
    outer_step,
    work_dir=None,
    updated_at=None,
    metadata_extra=None,
) -> GlobalMetadata
```

Publication sequence:

1. Save full state as local `latest.safetensors`.
2. Upload/overwrite `global/latest.safetensors`.
3. Construct metadata.
4. Write/upload `global/step_count.json`.
5. Return metadata.

Example:

```
{
  "algorithm": "dumb_diloco",
  "model_file": "global/latest.safetensors",
  "optimizer": "nesterov_sgd",
  "outer_step": 7,
  "updated_at": "2026-01-01T12:00:00+00:00"
}
```

> **⚠ NON-ATOMIC FIXED PATHS**
>
> The model and metadata are separate overwrites of fixed paths. A reader can observe old metadata with a newly uploaded model. A metadata failure can leave new weights advertised under the old outer step.

`download_global(local_path, metadata=None)`

Uses `metadata.model_file` without an allow-list. Repository write access can redirect workers to another repo file.

Delta publication

```
upload_delta(
    state,
    *,
    node_id,
    local_step,
    base_outer_step,
    work_dir=None,
    metadata=None,
) -> str
```

Remote path:

```
nodes/<node_id>/delta_<local_step>.safetensors
```

Safetensors metadata values are strings:

```
{
  "algorithm": "dumb_diloco",
  "base_outer_step": "7",
  "local_step": "500",
  "node_id": "worker-1"
}
```

The outer loop derives node/local step from the path and reads `base_outer_step` from the header.

`delta_paths()`

Returns all repository paths that start with `nodes/` and end in `.safetensors`. Parsing applies stricter path rules later.

Local cache layout

```
<cache_dir>/
├── downloads/<remote_path>
├── json/global/step_count.json
├── outgoing/global/step_count.json
└── speedtronic-download-<random>/   # temporary
```

The cache is not node-scoped and is not automatically rooted under run output.

Failure behavior

All `_retry` failures become `HubUnavailable`. Higher coordinator layers usually catch and log those errors, allowing local training to continue. Because permanent errors are retried, an invalid token or repository name can consume the full backoff schedule on every operation.

Security recommendations

- Use the SDK environment credential flow.
- Verify repository visibility after creation.
- Use one token with the minimum required scope.
- Restrict writer accounts.
- Do not treat `model_file` metadata as trusted input from an adversarial writer.
- Protect cache files from other local users.
- Monitor unexpected remote paths and updates.

Related: Overview, Outer Loop, and Security.

Outer optimizer and master loop

`NesterovOuterOptimizer`

```
NesterovOuterOptimizer(
    state,
    lr,
    momentum=0.9,
)
```

Maintains CPU FP32 momentum buffers for floating/complex state keys.

`step(state, delta)`

For every delta key:

1. Require the key to exist in global state.
2. Require matching shape.
3. Skip non-floating/non-complex state.
4. Convert gradient to CPU FP32.
5. Update momentum:

```
buffer = momentum * buffer + gradient
```

6. Compute:

```
effective = gradient + momentum * buffer
```

7. Update in the global state's native dtype:

```
state -= lr * effective
```

This matches PyTorch's Nesterov SGD formulation used by the repository test.

`state_dict()`

Returns cloned CPU momentum tensors only. Learning rate and momentum coefficient are not stored.

`load_state_dict(state)`

Replaces momentum buffers with cloned CPU FP32 tensors.

`OuterState`

A dataclass exists:

```
OuterState(
    global_state,
    outer_step=0,
    processed_deltas=set(),
    momentum={},
    last_error=None,
)
```

The current `MasterOuterLoop` stores equivalent fields directly and does not use this dataclass.

`MasterOuterLoop`

```
MasterOuterLoop(
    hub,
    state,
    *,
    node_state_dir,
    outer_lr=0.7,
    outer_momentum=0.9,
    poll_interval=60.0,
    on_global_update=None,
    logger=None,
)
```

Immediately clones the initial full state to CPU. Loading `outer_state.pt` replaces only matching same-shape tensors.

Properties:

```
loop.processed_filenames
loop.state_path
loop.metadata_path
```

Processed identity

```
(node_id, local_step, base_outer_step)
```

Node and local step come from the path; base step comes from safetensors metadata.

`processed_filenames` reconstructs only node/local paths and cannot distinguish different base-step metadata for the same file.

Candidate discovery

`_delta_candidates()` calls `hub.delta_paths()`, accepts paths parsed as:

```
nodes/<node>/delta_<integer>.safetensors
```

and sorts them by node, local step, and full path.

Delta download cache

Downloads are cached under:

```
<state_dir>/cache/deltas/<basename>
```

The basename omits the node ID, so two nodes with the same local step share a local path. Current code downloads immediately before reading, limiting immediate collision impact, but the cache is not safely node-namespaced.

One `sync_once()` round

Under one outer lock:

1. List and sort candidates.
2. Download every candidate.
3. Parse base outer step and construct identity.
4. Skip already processed identities.
5. Compute the expected floating/complex key set.
6. Require exact delta key-set equality.
7. Require matching shapes.
8. Log and skip unreadable/corrupt/incompatible files.
9. If no valid delta remains, emit an event and return.
10. Average all valid deltas in FP32.
11. Snapshot old global state, processed set, step, and momentum.
12. Apply one Nesterov outer update.
13. Increment outer step.
14. Add valid identities to the processed set.

15. Publish global state and metadata.
16. Save local outer state.
17. On publication/save failure, restore all in-memory snapshots and re-raise.
18. Invoke `on_global_update` with a cloned full state.
19. Emit `outer_step`.

The lock covers all network I/O, so snapshots from the training thread can block for a full round.

Validation

A delta is accepted when:

- safetensors and metadata load;
- keys exactly match all floating/complex global-state keys;
- each shape matches.

The implementation does not validate:

- NaN/infinity;
- magnitude;
- staleness against the current outer step;
- path/header node identity agreement;
- algorithm marker;
- model/config hash;
- dtype compatibility;
- cryptographic publisher identity.

Average

```
average = sum(valid_deltas) / len(valid_deltas)
```

Every delta has equal weight. There is no participant, sample, token, age, or quality weighting.

In-process rollback

If publishing or local persistence fails after in-memory mutation, the loop restores:

```
global_state
outer_step
processed_deltas
Nesterov momentum
```

The remote publication may already have succeeded before a later local failure, so the rollback cannot undo remote side effects.

Local master state

`outer_state.pt`:

```
global_state
outer_step
processed_deltas
momentum
```

`outer_metadata.json`:

```
outer_step
processed_deltas
updated_at
```

Both are written atomically through local temp-and-rename helpers. The JSON is diagnostic; the `.pt` file is authoritative.

Remote recovery

`adopt_global_state()` installs a complete remote state and outer step. It can reset momentum. It does not learn the remote processed-delta ledger or Nesterov momentum.

A hard crash after remote publication but before local state persistence can cause the restarted master to aggregate already-published deltas again.

Snapshot API

`snapshot()`

Under the lock, returns the outer step and a clone of the complete global state.

`start()`

Starts one daemon thread named `speedtronic-diloco-master` unless one is already alive.

Background loop

The thread calls `sync_once()` immediately, waits `poll_interval`, and repeats. Exceptions are logged and retried on the next interval.

`stop()`

Sets an event and joins for at most 10 seconds. It does not force a final synchronization and can return while network work is still running.

`state_dict()` and `load_state_dict()`

`state_dict()` returns:

```
outer_step
processed_deltas
momentum
global_state
```

`load_state_dict()` restores these values and writes the local state file as a side effect.

Unbounded growth

The remote delta tree, processed ledger, per-version global cache, and redownload workload grow over time. There is no delta deletion, compaction, or protocol-level processed list.

Multiple masters

No leader election, lease, or compare-and-swap protects global files. Multiple masters maintain independent momentum/processed sets and can overwrite the same paths with conflicting versions.

Related: Overview, Hub Transport, and Tensor Format.

Tensor and wire format

`cpu_tensor(value)`

```
value.detach().to(device="cpu").contiguous().clone()
```

This removes autograd history, transfers to CPU, makes contiguous layout, and clones storage. Cloning allows tied embedding/LM-head parameters to be serialized together, which safetensors rejects when they share memory.

`cpu_state_dict(state)`

Applies `cpu_tensor()` to every value and stringifies keys.

`save_safetensors(state, path, metadata=None)`

1. Creates parent directories.
2. Clones/normalizes every tensor.
3. Calls `safetensors.torch.save_file`.
4. Writes string metadata when supplied.

Integer and boolean state can be saved in global model files, but deltas exclude them.

`load_safetensors(path)`

Calls `safetensors.torch.load_file(path, device="cpu")`.

`read_metadata(path)`

Uses `safetensors.safe_open` to return safetensors header metadata as a string dictionary.

`floating_state(state)`

Returns keys whose tensors are floating-point or complex.

`compute_pseudo_gradient(baseline, current)`

Requirements:

- exact key-set equality;
- exact shape equality.

Output:

```
float32(baseline) - float32(current)
```

for floating/complex keys. Non-floating keys are omitted. Any key or shape mismatch rejects the entire delta.

The sign is intentionally baseline minus current: local optimization moves from baseline toward current, so this is the outer direction to add to global weights.

`average_deltas(deltas)`

- Rejects an empty list.
- Requires identical key sets.
- Uses the first delta's shapes as references.
- Accumulates in FP32.
- Divides by the number of deltas.

Dictionary key order is not significant.

`parse_delta_path(path)`

Accepts exactly:

```
nodes/<node_id>/delta_<integer>.safetensors
```

It splits on `/`, requires three segments, and parses the filename after removing `delta_` and `.safetensors`.

It does not validate that the file's embedded node ID and local step match the path.

`load_delta(path)`

1. Reads safetensors metadata.
2. Loads tensor state.
3. Converts tensor-loading failures to `ValueError` with a stable message.

The outer loop catches that error, logs the filename, and skips the candidate.

`metadata_json(metadata)`

Attempts to JSON-decode every metadata string. If any value is not valid JSON, it returns the original string dictionary. The current outer loop does not use this helper.

Global model file

`global/latest.safetensors` contains the complete cloned `state_dict`:

- trainable parameters;
- floating buffers;
- integer/boolean buffers;
- other registered state entries.

No safetensors metadata accompanies the global model. Version information is in `global/step_count.json`.

Global metadata file

```
{
  "algorithm": "dumb_diloco",
  "model_file": "global/latest.safetensors",
  "optimizer": "nesterov_sgd",
  "outer_step": 12,
  "updated_at": "2026-01-01T12:00:00+00:00"
}
```

Bootstrap metadata adds:

```
{"bootstrap": "true"}
```

Extra values are converted to strings.

Node delta file

Path:

```
nodes/<node_id>/delta_<local_step>.safetensors
```

Header:

```
{
  "algorithm": "dumb_diloco",
  "base_outer_step": "11",
  "local_step": "500",
  "node_id": "worker-1"
}
```

Tensor payload:

```
baseline - current
```

in FP32 for floating/complex state keys.

Full remote tree

```
<repo_id>/
├── global/
│   ├── latest.safetensors
│   └── step_count.json
└── nodes/
    ├── master-1/
    │   └── delta_500.safetensors
    ├── worker-1/
    │   ├── delta_500.safetensors
    │   └── delta_1000.safetensors
    └── worker-2/
        └── delta_500.safetensors
```

Local coordinator tree

```
<state_root>/<node_id>/
├── global/
│   └── latest-<outer_step>.safetensors
├── outgoing/
│   ├── latest.safetensors
│   └── <node_id>_delta_<step>.safetensors
└── cache/
    └── deltas/
        └── delta_<step>.safetensors
```

Master additionally stores:

```
outer_state.pt
outer_metadata.json
```

Compatibility rules

Model installation

Global installation is permissive:

- matching key/shape entries load;
- missing local/global keys are tolerated;
- shape mismatches are silently ignored.

Delta validation

Delta aggregation is strict:

- floating key set must match exactly;
- every shape must match;
- one bad delta is skipped in full.

There is no architecture/configuration fingerprint.

Non-floating state

Non-floating entries:

- are published in global model files;
- are installed on workers;
- do not appear in deltas;
- do not receive outer updates.

Models with running integer counters or model-specific non-floating synchronization semantics need custom handling.

Complex tensors

Complex keys pass floating-state checks, but arithmetic converts to FP32. Imaginary components are not preserved as true complex optimization. Real-valued models are the safe path.

Safetensors safety

Safetensors protects the tensor container from arbitrary pickle code execution, but it does not authenticate writers, encrypt data, or validate numerical values.

Related: Hub Transport, Outer Loop, and Security.

Shipped configurations

`configs/smoke.yaml`

A small, runnable, synthetic CPU-oriented run.

```yaml
name: smoke
seed: 1234
max_steps: 4
output_dir: runs/smoke
```

Top-level aliases populate `run`.

Setting	Effect
`name: smoke`	Descriptive run name
`seed: 1234`	Seeds global RNGs and synthetic data
`max_steps: 4`	Four absolute optimizer steps
`output_dir: runs/smoke`	Checkpoints resolve to `runs/smoke/checkpoints`

Model

```yaml
model:
  name: reference_transformer
  vocab_size: 128
  max_seq_len: 32
  n_layer: 2
  n_head: 4
  n_kv_head: 2
  d_model: 64
  d_ff: 128
```

The model has:

- 2 decoder blocks;
- 4 query heads;
- 2 KV heads with a repeat factor of 2;
- hidden width 64 and head width 16;

- vocabulary 128;
- context length 32;
- tied embeddings by default;
- zero dropout by default;
- gradient checkpointing disabled.

The bundled model contains 66,880 parameters in this configuration.

Data

```
data:
  synthetic: true
  num_tokens: 256
  block_size: 32
  micro_batch_size: 1
  target_batch_size: 2
  num_workers: 0
```

This creates 256 synthetic samples, each with 32 input tokens and 32 labels. Each optimizer update consumes two microbatches.

Optimizer and scheduler

```
optimizer:
  lr: 0.0003
  weight_decay: 0.01
scheduler:
  warmup_steps: 1
  max_steps: 4
```

Other defaults remain:

```
betas=(0.9, 0.95)
eps=1e-8
scheduler=cosine
min_lr_ratio=0.1
```

Precision and compilation

```
precision:
  mode: fp32
compile: false
```

This is the safest CPU configuration and disables compile overhead/fallback paths.

Checkpointing

```yaml
checkpoint:
  enabled: true
  directory: checkpoints
  every_steps: 2
  keep_last: 2
```

With the output base, the effective directory is:

```
runs/smoke/checkpoints
```

Saves occur at steps 2 and 4.

Logging

```yaml
logging:
  level: INFO
  every_steps: 1
```

Events go to stdout every step.

Run it

```
speedtronic train --config configs/smoke.yaml --device cpu
```

This configuration is the recommended local smoke test.

`configs/v2_smoke.yaml`

A CPU-safe run that exercises the v2 optimizer, cautious wrapper, shape profile, OOO fallback, and asynchronous distributed defaults without Hub access.

```yaml
optimizer:
  name: muon
  muon_plus: true
  cautious: true
shape_validation:
  enabled: true
  alignment: auto
ooo_backprop: true
ooo_streams: 2
distributed:
  enabled: false
  async_delta_upload: true
  async_global_poll: true
```

On CPU, `ooo_backprop` intentionally follows the standard sequential path. The Muon/AdamW hybrid, Muon+ normalization, cautious masking, and shape validation still run.

`configs/diloco.yaml`

A template for a larger local text run with distributed fields present but disabled.

> ⚠ **NOT RUNNABLE AS SHIPPED**
>
> It references `data/train.txt`, which is not included in the repository. Create or change that path before training. `speedtronic validate` performs schema validation and does not open the file.

Run

```yaml
name: local-demo
seed: 1234
max_steps: 1000
output_dir: runs/local-demo
```

This requests 1000 local optimizer steps.

Model

```yaml
model:
  name: reference_transformer
  vocab_size: 512
  max_seq_len: 128
  n_layer: 4
  n_head: 8
  n_kv_head: 2
  d_model: 256
  d_ff: 768
```

Architecture:

- 4 layers;
- 8 query heads;
- 2 KV heads;
- hidden width 256;
- head width 32;
- context 128;
- FFN configured as 768;
- vocabulary 512.

Text data

```yaml
data:
  text_path: data/train.txt
  block_size: 128
  micro_batch_size: 1
  target_batch_size: 8
  num_workers: 2
  prefetch_factor: 2
  pin_memory: true
```

This requests eight-way accumulation.

> **WORKER I/O**
>
> `TextFileTokenDataset` assigns disjoint blocks to workers, but each worker still reads the source file to discover them. Use a pre-sharded dataset for very large files.

Optimization

```yaml
optimizer:
  lr: 0.0003
  weight_decay: 0.1
scheduler:
  warmup_steps: 100
  max_steps: 1000
```

This uses 100 warmup optimizer steps and a 1000-step cosine horizon.

Performance features

```yaml
precision:
  mode: auto
compile: true
gradient_checkpointing: true
```

Auto precision chooses by device. Compilation and activation checkpointing are best-effort.

Disabled distributed section

```yaml
distributed:
  enabled: false
  mode: dumb_diloco
  role: single
  node_id: local
  inner_steps: 500
  poll_interval: 60
  outer_lr: 0.7
  outer_momentum: 0.9
  repo_id: your-org/your-run
  state_dir: .speedtronic/diloco
```

No coordinator is created while `enabled` is false. The values are a template for later enablement and must be replaced with a real unique repository/node configuration.

Checkpoints and logs

```yaml
checkpoint:
  enabled: true
  directory: checkpoints
  every_steps: 500
  keep_last: 3
logging:
  level: INFO
  file: runs/local-demo/train.log
  every_steps: 10
```

Effective checkpoint directory:

```
runs/local-demo/checkpoints
```

The log path is not rebased under output; it is already explicitly rooted at `runs/local-demo/train.log` relative to the process working directory.

Prepare and run

Create:

```
data/train.txt
```

or change `text_path` to an absolute path, then run:

```
speedtronic train --config configs/diloco.yaml
```

Configuration comparison

Property	`smoke.yaml`	`v2_smoke.yaml`	`diloco.yaml`
Runnable as shipped	Yes	Yes	No, missing text path
Steps	4	2	1000
Data	Synthetic	Synthetic	Text stream
Layers	2	1	4
Hidden width	64	32	256
Accumulation	2	2	8
Workers	0	0	2
Precision	FP32	FP32	Auto
Optimizer	AdamW	Muon + cautious	AdamW template
Compile	No	No	Yes
OOO backprop	Off	Enabled, CPU fallback	Off
Gradient checkpointing	No	No	Yes
Distributed	Disabled	Disabled, async defaults	Disabled template

Related: Configuration, Quickstart, and Shipped Examples.

Shipped examples

`examples/train_reference.py`

A programmatic CPU FP32 run on synthetic data.

```
from speedtronic.runtime import train_from_config

if __name__ == "__main__":
    result = train_from_config(
        {
            "run": {
                "name": "reference-example",
                "max_steps": 10,
                "device": "cpu",
                "output_dir": "runs/reference-example",
            },
            "model": {
                "name": "reference_transformer",
                "vocab_size": 128,
                "max_seq_len": 32,
                "n_layer": 2,
                "n_head": 4,
                "n_kv_head": 2,
                "d_model": 64,
                "d_ff": 128,
            },
            "data": {
                "block_size": 32,
                "num_tokens": 256,
                "micro_batch_size": 1,
                "target_batch_size": 2,
            },
            "precision": {"mode": "fp32"},
            "checkpoint": {
                "enabled": True,
                "directory": "checkpoints",
                "every_steps": 5,
            },
        }
    )
    print(result)
```

What it configures

Area	Value
Target	10 global steps
Device	CPU
Model	2-layer, 64-wide reference transformer
Data	256 synthetic samples of block size 32
Accumulation	2 microbatches per update
Precision	FP32
Checkpoint	Every 5 steps
Scheduler	Implicit default 1000-step cosine horizon

> **SCHEDULER HORIZON**
>
> The run stops at 10 steps while the scheduler still has a 1000-step horizon. Add an explicit scheduler section to align the schedule.

Run it

From the repository root:

```
python examples/train_reference.py
```

Artifacts:

```
runs/reference-example/checkpoints/
├── latest.json
├── step_000000000005.pt
└── step_000000000010.pt
```

The final dataclass output includes:

```
TrainResult(
  steps=10,
  samples=20,
  tokens=640,
  final_loss=<value>,
  elapsed_s=<value>,
  metrics=[...10 records...],
)
```

Exact loss and elapsed time vary.

`examples/diloco_master.yaml`

A Hub-backed master example using synthetic data.

> **⚠ EXTERNAL DEPENDENCY**
>
> `repo_id` is a placeholder. Running this example contacts Hugging Face Hub and requires credentials. It is not an offline or automatically runnable test.

Run and model

```yaml
name: diloco-example
max_steps: 10000
output_dir: runs/diloco-master
```

The local master trains for 10,000 optimizer steps.

Model:

```yaml
model:
  name: reference_transformer
  vocab_size: 512
  max_seq_len: 128
  n_layer: 2
  n_head: 8
  n_kv_head: 2
  d_model: 256
  d_ff: 768
```

Synthetic data

```yaml
data:
  block_size: 128
  num_tokens: 100000
  micro_batch_size: 1
  target_batch_size: 8
```

The field creates 100,000 synthetic samples, not a literal 100,000-token budget.

Optimizer and scheduler

```yaml
optimizer:
  lr: 0.0003
scheduler:
  warmup_steps: 100
  max_steps: 10000
```

Distributed section

```yaml
distributed:
  enabled: true
  mode: dumb_diloco
  role: master
  node_id: master-1
  collaborators: []
  repo_id: your-org/your-diloco-run
  token: null
  inner_steps: 500
  poll_interval: 60
  outer_lr: 0.7
  outer_momentum: 0.9
  state_dir: .speedtronic/diloco
```

Before running:

1. Replace `repo_id`.
2. Set `HF_TOKEN` in the environment.
3. Verify repository privacy.
4. Add only trusted collaborator usernames.
5. Keep `node_id: master-1` unique to this master.
6. Ensure all workers use identical model dimensions and compatible state keys.

Run:

```
export HF_TOKEN='...'
speedtronic train --config examples/diloco_master.yaml
```

Do not put the token in the YAML file.

Checkpoints

```yaml
checkpoint:
  enabled: true
  directory: checkpoints
  every_steps: 500
  keep_last: 3
```

The master checkpoint can include its local trainer state plus a second full global state and momentum, making files large.

Logging

```yaml
logging:
  level: INFO
  file: runs/diloco-master/train.log
  every_steps: 10
```

The master outer thread can also emit `outer_step` events independently of the training-step cadence.

No shipped worker configuration

The repository contains a master YAML but no dedicated worker YAML. Copy the configuration, change:

```yaml
role: worker
node_id: worker-1
```

and use the same model, data contract, and repository.

Example limitations

- Neither example is a test of real Hub failure recovery.
- The Python example inherits the scheduler-horizon mismatch.
- The master example is a placeholder and can create a real repository.
- Examples do not demonstrate data sharding across nodes.
- Examples do not demonstrate custom metric hooks.

Related: DumbDiLoCo, Shipped Configs, and Security.

Testing and development

Development install

```
cd /path/to/speedtronic
python3 -m venv .venv
. .venv/bin/activate
python -m pip install -e '.[dev]'
```

The development extra installs:

```
pytest>=7.4
ruff>=0.5
tomli>=2.0 on Python 3.10
```

Logging extras are separate.

Prescribed checks

```
python -m pytest -q
python -m ruff check src tests examples
python -m ruff format --check src tests examples
python -m compileall src
```

Pytest uses the `tests/` directory and `-ra`. Ruff targets Python 3.10 and a 100-character line length.

Test philosophy

The suite is CPU-only and uses fake Hub behavior where possible. It avoids network credentials and real accelerator requirements.

The current repository defines the v1 suite plus the v2 optimizer, systems, and asynchronous distributed tests. The generated inventory is the authoritative count.

Test inventory

`tests/test_config.py`

Test	Verifies
`test_config_yaml_aliases_and_accumulation`	Top-level run aliases, accumulation derivation, compile mapping
`test_config_round_trip`	Redacted save/load and equality
`test_unknown_keys_and_invalid_batch`	Unknown root rejection and invalid batch divisibility
`test_distributed_role_inference`	Enabled repo-bearing config defaults to master
`test_secret_can_be_redacted_for_serialization`	Explicit serialization redaction and redacted file save

`tests/test_model.py`

Test	Verifies
`test_reference_transformer_forward_and_loss`	Logit/loss shape and backward gradients
`test_reference_transformer_weight_tying_and_checkpoint_hook`	Shared weights and checkpoint flag propagation
`test_config_dimensions`	Default FFN width

`tests/test_precision_data.py`

Test	Verifies
`test_cpu_precision_defaults_to_fp32`	CPU auto FP32 and explicit CPU FP16 fallback
`test_streaming_text_dataset_and_tuple_collate`	Text block emission and tuple collation

`tests/test_training.py`

Test	Verifies
`test_trainer_runs_with_accumulation_and_checkpoint`	Three CPU steps, metric count, and checkpoint pointer
`test_accumulation_scales_gradients_and_reports_mean_loss`	Mathematical gradient accumulation and mean loss
`test_runtime_builds_reference_model`	Runtime composition and one reference-model step

`tests/test_distributed.py`

Test	Verifies
`test_pseudo_gradient_direction_and_average`	Baseline-current sign and equal averaging
`test_nesterov_outer_optimizer`	First Nesterov update formula and state shape
`test_delta_path_parser`	Valid and invalid delta paths
`test_tied_model_state_can_be_safetensors_serialized`	Clone behavior for tied storage

`tests/test_hub.py`

Test	Verifies
`test_hub_transport_round_trip`	Fake global/delta upload-download and collaborator grant
`test_master_skips_corrupt_delta_and_persists_processed_set`	Corrupt input is skipped and no state file is written

`tests/test_v2_optimizers.py`

Covers v2 config aliases, Newton–Schulz edge cases, Muon+ normalization, role-aware routing, hybrid/AdamW construction, closure forwarding, cautious masking, sparse rejection, and a CPU runtime smoke build.

`tests/test_v2_systems.py`

Covers shape profiles and warning suggestions, standard-library logger compatibility, causal label alignment, scheduler defaults, and CPU OOO fallback.

`tests/test_v2_distributed.py`

Covers asynchronous global polling, non-blocking delta dispatch, queue overflow, prepared resume, synchronous transport, and baseline retention on failure.

`tests/test_v2_release.py`

Covers version synchronization, CLI redaction, and serialization of all v2 configuration sections.

Coverage matrix

Area	Direct coverage	Important gaps
Config basics	Good scalar/alias/redaction coverage	Rare nested types and hardware/model compatibility
Trainer accumulation	Good CPU scalar/list coverage	Resume equivalence, output-contract breadth
Reference model	Basic forward/loss	RoPE, GQA, masks, causality, eval mode
CPU precision	Partial	CUDA, MPS, fused, scaler resume
Data	Basic plus worker sharding	Prefetch, custom tokenizer, loader builder
Checkpoints	Basic existence and final-step save	Contents, retention, corruption, RNG
Hub transport	Fake happy path	Retries, real SDK variants, cache fallback
Outer optimizer	Basic formula and publish-path tests	Complex values and full recovery matrix
Outer loop	Corrupt skip and lock-scope tests	Real Hub aggregation and crash recovery
Coordinator	Async dispatch, poll, resume, stop coverage	Master/worker real Hub lifecycle
CLI	Validation redaction and exit code	Parsing, overrides, train error mapping
Logging/hooks	None	Cadence, JSONL, W&B, TensorBoard
Compile/checkpoint fallback	None	Trainer fallback behavior
Packaging/docs	Version, wheel, Twine, docs build gates	Platform matrix and CUDA runners

v2 verification

The v2 tests cover:

- scalar and mapping optimizer configuration;
- Muon routing, tied weights, Newton–Schulz, Muon+, and cautious wrapping;
- sparse/non-matrix rejection;

- CPU shape profiles and warning suggestions;
- causal label alignment and scheduler synchronization;
- CPU no-op OOO behavior;
- asynchronous delta dispatch, queue overflow, failure, and baseline retention;
- synchronous transport compatibility.

CUDA stream parity, fused AdamW, and hardware-specific speedups require a CUDA runner and are not implied by the CPU suite.

Documentation coverage build

The Docusaurus project has a separate coverage gate:

```
cd docs-site
npm ci
npm run build
```

Before compilation it:

1. parses every local source module with Python's AST;
2. generates one inventory page for all modules, classes, functions, methods, tests, configs, and examples;
3. validates inventory paths and symbol anchors;
4. validates sidebar coverage.

Docusaurus then fails on broken internal links and MDX issues.

Adding a source module

When adding a Python module under `src/speedtronic`:

1. Curate its conceptual responsibility in an appropriate page.
2. Ensure the generated inventory includes all declarations.
3. Add or update tests.
4. Add a link from the module coverage page if needed.
5. Run all Python and docs checks.

The generator requires no manual inventory edits; it rewrites the generated page.

Code style

The repository uses:

- Ruff `E`, `F`, and `I` rules;
- 100-character lines;
- `from __future__ import annotations`;
- dataclasses for configuration and results;
- explicit fallback logging for optional capabilities.

Test-data safety

Do not add real credentials to configs or tests. The fake Hub API stores bytes in memory and does not use network access.

Related: Test Map, Generated Source Inventory, and Troubleshooting.

Packaging and release

Distribution identity

```
name: speedtronic
version: 2.0.0
requires-python: >=3.10
license: Apache-2.0
```

The distribution and import package are both named `speedtronic`. The repository URL may use title case, but PyPI normalization and metadata are exactly `speedtronic`.

Build system

```
[build-system]
requires = ["setuptools>=77.0.3", "wheel"]
build-backend = "setuptools.build_meta"
```

The modern SPDX license expression and `license-files = ["LICENSE"]` require a sufficiently recent setuptools backend.

Runtime dependencies

```
torch>=2.1
safetensors>=0.4
PyYAML>=6.0
numpy>=1.24
huggingface-hub>=0.23
```

Install the appropriate CPU or CUDA PyTorch wheel for the target environment when the default PyPI wheel is not appropriate.

Optional dependencies

Development

```
pytest>=7.4
ruff>=0.5
```

Logging

```
wandb>=0.16
tensorboard>=2.14
```

Release

```
build>=1.2
check-wheel-contents>=0.6
twine>=6.0
```

The release extra is for maintainers and is not required at runtime.

Console entry point

```
[project.scripts]
speedtronic = "speedtronic.cli:main"
```

This creates:

```
speedtronic ...
```

The package also supports:

```
python -m speedtronic ...
```

Source layout

```
[tool.setuptools]
package-dir = {"" = "src"}

[tool.setuptools.packages.find]
where = ["src"]
include = ["speedtronic*"]
namespaces = false
```

The wheel contains only regular packages matching `speedtronic*`:

```
speedtronic/
speedtronic/distributed/
```

Tests, configs, and examples are not installed as wheel package data.

Source distribution manifest

`MANIFEST.in` includes:

```
LICENSE
README.md
pyproject.toml
configs/*.yaml
examples/*.py
examples/*.yaml
CHANGELOG.md
tests/*.py
```

It excludes bytecode and generated run/state directories.

Wheel versus sdist

Artifact	Intended use	Includes
Wheel	Installation	Importable package, metadata, license, entry point
Source archive	Source review/build	Wheel source plus tests, configs, examples, manifest, README, license

Commands such as `speedtronic train --config configs/smoke.yaml` are source-checkout/source-archive examples. An arbitrary installed wheel does not guarantee that top-level `configs/` exists beside the environment.

Version synchronization

The version appears in:

```
pyproject.toml
src/speedtronic/__init__.py
```

They currently both contain `2.0.0`. A release should update both and rebuild.

Local release sequence

```
python -m pip install -e '.[dev]'
python -m pytest -q
python -m ruff check src tests examples
python -m ruff format --check src tests examples
python -m compileall src

python -m pip install build twine check-wheel-contents
rm -rf dist build
python -m build
python -m twine check dist/*
check-wheel-contents dist/speedtronic-2.0.0-py3-none-any.whl
```

Install into a clean environment and test the console command before upload.

Artifact inspection

For a wheel:

- verify `Name: speedtronic`;
- verify version and Python requirement;
- verify Apache license file;
- verify console entry point;
- verify `speedtronic` and `speedtronic.distributed` modules;
- reject unexpected top-level packages.

For an sdist:

- inspect `SOURCES.txt`/archive contents;
- ensure tests/configs/examples are included;
- ensure virtual environments, runs, checkpoints, and bytecode are absent.

PyPI upload

The project is published at:

```
https://pypi.org/project/speedtronic/
```

Do not print or inspect credential files. Let Twine resolve its existing non-interactive configuration or SDK credential environment. A PyPI release filename cannot be overwritten once uploaded; fix metadata and increment the version for a new release.

Documentation build

The Docusaurus project is isolated under `docs-site` and does not affect the Python wheel.

```
cd docs-site
npm ci
npm run build
npm run serve
```

Generated directories:

```
docs-site/node_modules/
docs-site/.docusaurus/
docs-site/build/
```

are ignored locally.

Release checklist

- [ ] Version values agree.
- [ ] Changelog/release intent is defined outside the source if needed.
- [ ] Tests, lint, formatting, and compilation pass.
- [ ] Wheel and sdist build.
- [ ] Twine metadata check passes.
- [ ] Wheel contents are clean.
- [ ] Clean-environment installation works.
- [ ] CLI and smoke run work from the wheel/source archive.
- [ ] No secret is stored in source or distribution metadata.
- [ ] Documentation build passes without deployment.

Related: Testing, Shipped Examples, and Troubleshooting.

Troubleshooting

Configuration errors

`unknown configuration keys`

Speedtronic rejects unknown keys intentionally.

Check:

- section spelling;
- top-level aliases (`name`, `seed`, `device`, `max_steps`, `output_dir`, `log_every`);
- the current configuration schema;
- whether `logging.hooks` was added—hooks are programmatic in 2.0.0.

`batch sizes must be positive`

Set positive `micro_batch_size` and `target_batch_size`.

`target_batch_size must be ... divisible`

Use:

```
target_batch_size % micro_batch_size == 0
```

Model dimensions invalid

For the bundled model:

- `n_kv_head` must divide `n_head`;
- `d_model` must divide by `n_head`;
- attention head dimension must be even for RoPE;
- all dimensions must be positive.

Distributed repository missing

Enabled DumbDiLoCo requires:

```
distributed:
  enabled: true
  repo_id: org/run
```

Role changed unexpectedly

An enabled `single` role with a repository becomes `master`. Set `role: worker` explicitly on workers.

Validation passes but training fails

`validate` checks schema and selected invariants only. It does not open files, build models, resolve hardware, or run data.

Common structurally valid but unrunnable configurations:

- missing `data.text_path`;
- unknown custom model name;
- model block size smaller than data block size;
- data vocabulary inconsistent with model vocabulary;
- requested unavailable CUDA/MPS device.

Model errors

`unknown model`

A custom model registered in one process is not automatically visible in a CLI process. Construct the runtime in the same process or add an application plugin import.

`sequence length ... exceeds model block size`

The reference model rejects inputs longer than `model.max_seq_len`. Keep:

```
data.block_size <= model.max_seq_len
```

for the bundled model.

Unexpected `input_ids`, `labels`, or `attention_mask`

The trainer expands dictionary batches as keyword arguments. A custom model must accept the emitted keys or use a tuple batch/custom loader.

Loss behaves incorrectly

A bare tensor is treated as a direct loss. Return `{"logits": logits}` plus compatible labels or return a loss mapping when using 3-D output tensors.

Causal target appears shifted twice

Speedtronic 2.0 consumes the already-shifted labels emitted by its datasets. If a custom model or dataset still shifts the same labels, align the contract to one next-token label per input position.

Data errors

`data loader produced no batches`

The complete loader pass yielded no items. Check:

- empty iterable dataset;
- text file exists and is readable;
- final stream behavior;
- dataset filter removed all examples;
- `drop_last` with a dataset smaller than one batch.

`data.dataset must ... exist`

The configured serialized path is absent or is not a PyTorch Dataset.

Never load an untrusted `.pt` dataset; `weights_only=False` permits pickle code execution.

Duplicate text data with workers

The bundled text iterable dataset is not worker-sharded. Set `num_workers: 0` or implement worker-aware sharding.

Broken custom tokenizer

The stream carries IDs, not raw text. Context-sensitive tokenizers can split merges at chunk boundaries. Use an incremental tokenizer or a whole-file preprocessing step.

Out-of-range token IDs

The standard runtime passes `model.vocab_size` to the loader and ignores a separate `data.vocab_size`. Keep vocabularies aligned.

Runtime and precision errors

CUDA/MPS unavailable

`resolve_device()` raises when an explicit backend is unavailable. Use `--device cpu` or `auto`.

Unexpected FP32 fallback

Expected cases include:

- auto on CPU/MPS;
- explicit CPU FP16;
- mixed precision on MPS;
- unsupported CUDA BF16.

The `train_start` event reports the resolved mode.

CPU BF16 failure

Explicit CPU BF16 is not capability-checked. Use FP32 if the installed PyTorch/custom operators cannot run CPU autocast BF16.

Fused AdamW not used

Fused construction is best-effort and may fail because of device placement or backend support. No warning is emitted by the fallback path.

Compilation

Compilation disabled automatically

A compile event reports construction/runtime fallback. The run continues eagerly.

Model executes twice

The compiled forward raised, then the trainer reran the same batch eagerly. Look for model side effects or an actual model error masked by the broad fallback.

`torch.compile` overhead on short runs

Compilation cost can dominate a four-step smoke test. Keep `compile: false` for smoke runs.

Scheduler behavior

Run ends during warmup

Set `scheduler.max_steps` and `warmup_steps` explicitly. Programmatic run-target overrides do not reliably update the default 1000-step scheduler.

Logged LR is one step ahead

The optimizer steps before the scheduler advances, and metrics read the post-scheduler LR.

GradScaler behavior

On CUDA FP16, an overflow can cause `GradScaler` to skip the optimizer update while Speedtronic still increments the global step and scheduler. A sudden high loss or unchanged parameters around an early step can indicate this.

Checkpoint and resume

No checkpoint found

Resume searches:

```
<resolved output_dir>/<checkpoint.directory>/
```

Changing `--output-dir` changes the search location.

A warning is logged and a new run starts.

Checkpoint is stale behind file scan

If `latest.json` is missing, the manager chooses the highest valid filename. If the pointer is valid, it trusts the pointer.

Corrupt newest checkpoint

`load_latest()` does not fall back to an older file after a corrupt selected checkpoint. Restore a known-good copy or point the directory to one.

Resume repeats data

DataLoader position and worker state are not checkpointed. Expect approximate replay.

Final step not saved

Only exact checkpoint intervals save. A final off-interval step is lost from the latest checkpoint.

Checkpoint shape mismatch

Current config/model compatibility is not checked before `load_state_dict`. Ensure identical architecture and state keys.

Logging and integrations

No W&B/TensorBoard configuration

`LoggingConfig` has no hooks field. Attach adapters programmatically.

TensorBoard file remains open

Call `TensorboardHook.close()`.

W&B run does not finish

The adapter has no public close method. Finish the run through the Wandb API or application lifecycle.

JSONL has concurrent writes

The master outer thread and trainer can emit concurrently. The logger does not lock JSONL appends or hook invocation.

DumbDiLoCo

Worker cannot find global metadata

The master may not have bootstrapped yet, credentials may lack access, or the repository may be wrong. The worker continues locally and retries later.

Repeated slow retries

Every exception, including 401/403 and 404, is retried with backoff. Verify repository/token configuration before waiting through repeated delays.

Corrupt delta warnings

The master skips the file and retries it on later polls because invalid files are not marked processed. Remove or replace an unrecoverable remote file.

Delta key/shape mismatch

All nodes must use the same floating-state key set and shapes. A custom architecture or version skew rejects the full delta.

No global update

Check:

- delta was uploaded successfully;
- `outer_step` increased;
- `base_outer_step` and identities are valid;
- one active master is running;
- the worker is not stuck behind an invalid cache;
- model state shapes match.

Repository grows continuously

Expected in 2.0.0. Deltas are not deleted, global caches are not pruned, and all listed candidates are downloaded.

Unexpected optimizer reset

Optimizer state clears after a successful delta upload **or** global installation, not strictly at inner boundaries.

Resume produces a large delta

Current startup/load ordering can pair a fresh global model with a checkpointed older baseline. Treat distributed resume as a known limitation and verify baseline/global step before training.

v2 optimizer and systems issues

Muon routes too few parameters

Check the parameter role heuristic and tied weights. Embeddings, heads, norms, and biases intentionally remain on AdamW. A model can set `_speedtronic_optimizer_role` to make an explicit choice.

Muon fails on a sparse or complex gradient

Muon is defined for dense, real, floating-point 2-D matrices. Route sparse or complex parameters to AdamW or provide a custom optimizer.

OOO backprop does nothing

`ooo_backprop` is a no-op on CPU and MPS. It is also disabled when `compile: true`, when fewer than two module stages are found, or when CUDA streams cannot be installed. Inspect the `ooo_backprop` event.

Shape warnings appear on CPU

Automatic CPU profiles are quiet. An explicit `shape_validation.alignment` also needs `warn_on_cpu: true` to enable CPU benchmarking warnings. Set `alignment: none` to disable them.

Async delta is skipped

A `delta_upload_skipped` event with `reason=upload_in_flight` means the one-slot queue is occupied. The next accepted boundary sends a cumulative delta. Increase `delta_upload_shutdown_timeout` only for controlled shutdown tests, not to make the inner loop block.

Async poll does not install a new global immediately

Downloads are intentionally performed in the background and installed only at a training-thread boundary. Inspect `delta_upload` and `ooo_backprop`/`shape` events plus the outer-step metric.

Diagnostics

For local issues, collect:

```
resolved configuration
device and precision plan
start/target step
checkpoint pointer
first failing stack trace
metric records
exact model/data dimensions
node role/ID/repository when distributed
```

Never collect or print token values. Share redacted configuration and filenames instead.

Related: Security and Limitations, Configuration, and DumbDiLoCo.

Security and limitations

Security boundaries

Pickle-capable local files

The following paths use `torch.load(..., weights_only=False)` or equivalent pickle-capable loading:

- local trainer checkpoints;
- serialized datasets configured through `data.dataset`;
- master `outer_state.pt`.

Only load files from trusted local sources. A write-capable local attacker can craft a checkpoint that executes code during deserialization.

Hub writer trust

DumbDiLoCo assumes a closed set of trusted repository writers. A collaborator with write access can:

- replace global model weights;
- replace node deltas;
- alter step metadata;
- set `model_file` to another repository path;
- overwrite the fixed global paths.

There is no:

- node signature;
- content hash in metadata;
- pinned Hub revision;
- compare-and-swap;
- Byzantine quorum;
- outlier rejection;
- contribution-size limit;
- identity verification;
- public/open participation mode.

Secret handling

- `config.save()` and trainer checkpoints redact `distributed.token`.
- `to_dict()` and `to_yaml()` are unredacted by default when called programmatically.
- `speedtronic validate` redacts the distributed token in both JSON and YAML output.
- Prefer Hugging Face SDK environment authentication.

Never inspect or print credential files during diagnostics.

Repository privacy

The master requests `private=True` when creating a repository, but `exist_ok=True` does not verify that an existing same-name repository is private. Verify visibility explicitly.

Path trust

- `node_id` is used in local and remote paths.
- Explicit IDs are checked for `/`, `\\`, `.`, and `..` forms.
- Environment-derived IDs are not revalidated.
- `model_file` metadata is followed without an allow-list.
- Checkpoint `latest.json` file names are joined to the checkpoint directory without strict filename-pattern validation.

Denial of service

A malicious delta can use valid keys/shapes but extreme values. The master will average and apply it. Hub operations also have broad retries with no request timeout.

Known correctness and design issues

Causal label contract

Speedtronic 2.0 consumes the already-shifted labels emitted by its synthetic and text datasets. Custom models must follow the same one-next-token-label per input-position contract; shifting both dataset and model will skip a target.

Text dataset worker sharding

`TextFileTokenDataset` uses the PyTorch worker identity to yield disjoint blocks to each worker. Each worker still reads the source file to discover those blocks, so I/O scales with worker count; use a pre-sharded dataset for large files.

Stream-unsafe arbitrary tokenizers

The text stream carries token IDs rather than raw text across chunk boundaries. BPE/SentencePiece-style merges can be split.

Scheduler inference

The historical 1000-step scheduler default is synchronized to a shorter explicit `run.max_steps` target unless the scheduler horizon is set explicitly.

Incomplete resume

The checkpoint does not preserve DataLoader iterator, sampler, epoch, worker RNG, or persistent worker state. Resume can repeat or skip examples.

GradScaler step accounting

CUDA FP16 overflow can skip the optimizer update. The trainer emits `optimizer_step_skipped` and does not increment the step, scheduler, or coordinator boundary for that attempt; retries continue at the same step.

Fused AdamW device probe

The model is moved to the resolved device before optimizer construction, so fused-AdamW capability probing and the actual parameters share a device. A failed fused construction still falls back to ordinary AdamW.

Compile fallback breadth

Any compiled-forward exception disables compilation and reruns eagerly. A genuine model error can be hidden if eager execution succeeds; side effects/RNG can be duplicated.

Model training mode

The trainer explicitly calls `model.train()` before fitting. Custom `fit()` callers that bypass this path remain responsible for their mode.

Checkpoint pointer consistency

A checkpoint file can be written before `latest.json` is updated. A valid pointer to a corrupt file does not fall back to older checkpoints. The trainer forces one final checkpoint at the end of a successful fit when the configured interval did not land on the final step.

Data vocabulary override

The runtime always passes `model.vocab_size` to the loader, so a separately configured `data.vocab_size` is ignored.

Configuration semantics

Validation is intentionally shallow. It does not check model registration, data paths, hardware, or model/data compatibility. Boolean strings are coerced with `bool(...)`.

v2 limitations

- Muon is a research-backed optimizer whose convergence and simplicity-bias trade-offs are not fully characterized.
- Cautious masking observes the base optimizer's complete update, including decoupled weight decay, when wrapped around a third-party optimizer.
- OOO backprop is opt-in, CUDA-only in its active path, and mutually exclusive with `torch.compile`.
- Shape validation is advisory and does not guarantee a speedup.
- AoT scheduling is deferred as subsumed by `torch.compile`; no private scheduler API is shipped.
- Async Hub uploads are bounded to one in-flight job and may be skipped.
- Pending upload state increases checkpoint size by up to one model snapshot.

Distributed correctness limitations

Fixed-path publication is not atomic

Weights and metadata are uploaded separately. Readers can observe mismatched versions, and a failure between uploads can leave new weights under old metadata.

Local processed-delta ledger is not remote

A fresh master or a crash after remote publication but before local persistence can aggregate historical deltas again.

Multiple masters are unsupported

There is no leader election or distributed lock. Multiple masters overwrite the same global paths and maintain conflicting state.

Unbounded storage and bandwidth

Remote delta files, local processed sets, per-version global caches, and downloaded-delta caches grow indefinitely. Every poll downloads all listed candidates to inspect headers before dedupe.

Delta cache collisions

The master cache uses only `delta_<step>.safetensors` basenames, so nodes with the same local step share a local path.

Broad retry policy

Authentication, 404, malformed request, and programming errors are retried like transient outages. Hub work is dispatched to bounded background lanes by default, so a stuck request can delay a future boundary but does not block the ordinary local inner step.

Coordinator restart lifecycle

A stopped v2 coordinator resets its started flag so a later `fit()` can start fresh background lanes. A request that outlives the bounded shutdown timeout remains a daemon-side resource warning.

Resume baseline ordering

Coordinator startup stages checkpoint state before remote refresh when the v2 `prepare_state` path is available. A checkpointed model/baseline pair therefore wins at startup, and newer remote state is discovered by the background poll lane.

Partial inner-loop recovery

A worker restart begins from the latest global and can discard local progress since that publication. v2 restores one bounded pending upload job from a coordinator checkpoint, but it does not reconstruct every uncommitted inner optimizer step.

Reset semantics are broader than the name

Optimizer state clears after a successful upload or global load, including non-boundary global polls.

No distributed data sharding

Node ID, role, and rank do not affect data seed/order. Nodes using identical configurations can see identical synthetic samples.

Compatibility is shape-based

No model fingerprint, config hash, schema version, dtype migration, or cryptographic lineage is checked.

Non-floating buffers

Integer/boolean buffers are transported globally but excluded from deltas and outer optimization. Model-specific counter or running-state semantics are not handled.

Complex tensors

Complex state passes floating checks but arithmetic converts to float32, so imaginary values are not optimized as true complex tensors.

API and maintenance limitations

- YAML cannot register arbitrary model factories.
- YAML cannot configure metric hooks.
- Standard config fields are transformer-oriented.
- Model-returned auxiliary metrics are discarded by the trainer.
- The default causal dictionary collator is not general-purpose.
- `MetricLogger.close()` is not called by `train_from_config()`.
- `TrainResult.metrics` grows on every step.
- Compatibility aliases and declarative-but-unused types add surface area.
- Repository tests do not cover CLI, logging, coordinator lifecycle, successful outer aggregation, resume, or real Hub networking.

Performance limitations

- Per-microbatch loss-to-CPU transfer synchronizes CUDA.
- Attention-mask token counting synchronizes through `.item()`.
- Dense explicit attention masks can reduce SDPA efficiency.
- Fused AdamW may silently not engage.
- Compile and checkpoint costs are included in wall-clock step rates.
- Hub I/O is dispatched to bounded background lanes by default; synchronous mode remains available for deterministic debugging.

Explicit non-goals

Speedtronic 2.0.0 does not provide:

- FSDP;
- pipeline parallelism;
- tensor parallelism;
- a model zoo;
- tokenizer training;
- a hosted dashboard;
- open/adversarial distributed participation;
- a dedicated parameter server;
- exact mid-inner-loop worker resume;
- broad model/data schema plugin loading.

Recommended production posture

1. Treat version 2.0.0 as alpha.
2. Use trusted local checkpoint/dataset files only.
3. Add regression tests for any distributed deployment scenario.
4. Use a private Hub repository and one active master.
5. Grant write access only to trusted service accounts.
6. Explicitly set unique node IDs.
7. Protect state/cache directories with filesystem permissions.
8. Avoid YAML tokens and unredacted validation output.
9. Version Hub repositories per experiment.
10. Validate numerical behavior on target hardware before long runs.

Related: DumbDiLoCo, Checkpointing, and Troubleshooting.

Source coverage

Coverage claim

This documentation project inventories:

- all implementation modules under `src/speedtronic` (v1 plus v2);
- every top-level class and function;
- every direct class method;
- module constants and compatibility aliases;
- all v1 and v2 test functions in the repository;
- both shipped YAML configurations;
- both shipped examples;
- package metadata, source manifest, license, and ignore rules through the operational pages.

The exhaustive AST listing is generated during `npm run build` and appears as Generated Source Inventory.

Build-time coverage gate

`scripts/validate_coverage.py` fails the build when:

- a Python implementation file is absent from the generated inventory;
- a top-level class/function or class method lacks a generated anchor;
- a test file or test function is absent;
- a shipped config or example is absent;
- a documentation page is not represented in the sidebar.

The source inventory generator uses `ast.parse`; it does not import Speedtronic, load PyTorch, read credentials, or access the network.

Implementation module map

Module	Responsibility	Curated page
`__init__.py`	Eager/lazy public exports and version	Generated inventory
`__main__.py`	`python -m speedtronic`	CLI
`config.py`	Dataclasses, parsing, defaults, redaction	Configuration
`runtime.py`	Composition helpers	Runtime and Trainer
`trainer.py`	Optimization loop and state	Runtime and Trainer
`data.py`	Tokenizers, datasets, collators, loader	Data
`model.py`	Reference transformer	Reference Model
`registry.py`	Model factory registry	Extensions
`precision.py`	Device and precision capability	Precision
`checkpoint.py`	Atomic persistence and RNG	Checkpoints
`profiling.py`	Metrics and callbacks	Observability
`hooks.py`	Event wrapper	Observability
`integrations.py`	W&B and TensorBoard adapters	Observability
`module_utils.py`	Gradient-checkpointing helper	Extensions
`cli.py`	Argument parser and command dispatch	CLI
`distributed/__init__.py`	Distributed re-exports	DumbDiLoCo overview
`distributed/diloco.py`	Master/worker coordinator	Coordinator
`distributed/hub.py`	Hub file transport and retries	Hub Transport
`distributed/outer.py`	Outer optimizer and master loop	Outer Loop
`distributed/tensors.py`	Safetensors and delta wire format	Tensor Format

Module	Responsibility	Curated page
`optimizers.py`	Hybrid Muon, Muon+, cautious wrapper, routing	v2 Optimizers
`shapes.py`	Startup shape profiles and warnings	Shape Validation
`scheduling.py`	CUDA stage streams and OOO fallback	OOO Backprop

Supporting file map

File	Coverage
`README.md`	User overview reconciled against implementation in all pages
`pyproject.toml`	Packaging
`MANIFEST.in`	Packaging
`LICENSE`	Apache-2.0 summary and glossary
`.gitignore`	Generated/artifact guidance in operations pages
`configs/smoke.yaml`	Shipped Configs
`configs/v2_smoke.yaml`	Shipped Configs
`configs/diloco.yaml`	Shipped Configs
`CHANGELOG.md`	Release and deferred-decision record
`.github/workflows/ci.yml`	Python, documentation, and distribution gates
`examples/train_reference.py`	Shipped Examples
`examples/diloco_master.yaml`	Shipped Examples

Evidence labels

Label	Meaning
Verified by tests	Directly asserted by a repository test
Example exercised	Present in a shipped runnable configuration or Python example
Implementation-derived	Read directly from source but not directly tested
External dependent	Requires real CUDA, MPS, compilation, filesystem failure, or Hub networking

Documentation completeness does not imply every documented path is tested. The test map makes that distinction explicit.

Generated source inventory

> **INFO**
>
> This page is generated from the local repository by `scripts/generate_source_inventory.py` during every documentation build. It inventories every implementation module and every top-level class/function/method without importing PyTorch or contacting the network.

Implementation modules

`src/speedtronic/__init__.py`

Import path: `speedtronic`
Purpose: Speedtronic: fast, efficient training with optional DumbDiLoCo..
Lines: 108

Imported modules: `__future__.annotations`, `.config.CheckpointConfig`, `.config.Config`, `.config.ConfigError`, `.config.DataConfig`, `.config.DistributedConfig`, `.config.LoggingConfig`, `.config.ModelConfig`, `.config.OptimizerConfig`, `.config.PrecisionConfig`, `.config.RunConfig`, `.config.SchedulerConfig`, `.config.ShapeValidationConfig`, `.config.SpeedtronicConfig`, `.config.load_config`, `.config.load_yaml_config`, `.module_utils.set_gradient_checkpointing`, `.optimizers.CautiousOptimizer`, `.optimizers.HybridOptimizer`, `.optimizers.Muon`, `.optimizers.ParameterRouting`, `.optimizers.newton_schulz`, `.optimizers.post_polar_normalize`, `.optimizers.route_parameters`, `.registry.ModelRegistry`, `.registry.build_model`, `.registry.register_model`, `.registry.registry`, `.scheduling.StageInfo`, `.scheduling.StageStreamScheduler`, `.shapes.ShapeProfile`, `.shapes.ShapeReport`, `.shapes.ShapeWarning`, `.shapes.resolve_shape_profile`, `.shapes.validate_startup_shapes`

Function `__getattr__` {#api-speedtronic-getattr}

```
def __getattr__(name: str)
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	42
constant/alias	`__version__`	87

`src/speedtronic/__main__.py`

Import path: `speedtronic.__main__`
Purpose: No module docstring.
Lines: 4

Imported modules: `.cli.main`

`src/speedtronic/checkpoint.py`

Import path: `speedtronic.checkpoint`
Purpose: Atomic local checkpoint storage and discovery..
Lines: 176

Imported modules: `__future__.annotations`, `json`, `os`, `re`, `shutil`, `pathlib.Path`, `typing.Any`, `torch`

Class `CheckpointError`

```
class CheckpointError(RuntimeError)
```

Function `atomic_torch_save`

```
def atomic_torch_save(state: dict[str, Any], path: str | os.PathLike[str]) -> None
```

Function `atomic_json_dump`

```
def atomic_json_dump(value: Any, path: str | os.PathLike[str]) -> None
```

Class `CheckpointManager`

```
class CheckpointManager
```

Manage local `step_<n>.pt` checkpoints and a latest pointer.

Method `__init__` {#api-speedtronic-checkpoint-CheckpointManager-init}

```
def __init__(self, directory: str | os.PathLike[str], *, every_steps: int=500, keep_last: int | None=3, enabled: bool=True) -> None
```

Method `should_save`

```
def should_save(self, step: int) -> bool
```

Method `path_for`

```
def path_for(self, step: int) -> Path
```

Method `save`

```
def save(self, step: int, state: dict[str, Any]) -> Path
```

Method `load_latest`

```
def load_latest(self) -> dict[str, Any] \| None
```

Method `latest_path`

```
def latest_path(self) -> Path \| None
```

Method `_checkpoint_paths`

```
def _checkpoint_paths(self) -> list[Path]
```

Method `prune`

```
def prune(self) -> None
```

Method `copy_to`

```
def copy_to(self, destination: str | os.PathLike[str]) -> Path
```

Function `capture_rng_state`

```
def capture_rng_state() -> dict[str, Any]
```

Function `restore_rng_state`

```
def restore_rng_state(state: dict[str, Any] | None) -> None
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	169

`src/speedtronic/cli.py`

Import path: `speedtronic.cli`
Purpose: Command-line interface for Speedtronic..
Lines: 75

Imported modules: `__future__.annotations`, `argparse`, `json`, `sys`, `typing.Sequence`, `.config.ConfigError`, `.config.SpeedtronicConfig`

Function `_parser`

```
def _parser() -> argparse.ArgumentParser
```

Function `main`

```
def main(argv: Sequence[str] | None=None) -> int
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	75

`src/speedtronic/config.py`

Import path: `speedtronic.config`
Purpose: Configuration objects and loading helpers for Speedtronic. The configuration is intentionally data-first: a run can be represented by a YAML file or by constructing the same dataclasses in Python.
Lines: 631

Imported modules: `__future__.annotations`, `json`, `os`, `dataclasses.asdict`, `dataclasses.dataclass`, `dataclasses.field`, `dataclasses.fields`, `dataclasses.is_dataclass`, `pathlib.Path`, `typing.Any`, `typing.Mapping`, `typing.TypeVar`

Class `ConfigError`

```
class ConfigError(ValueError)
```

Raised when a run configuration is invalid.

Class `ModelConfig`

```
class ModelConfig
```

Method `__post_init__` {#api-speedtronic-config-ModelConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`name`	28
field	`vocab_size`	29
field	`max_seq_len`	30
field	`n_layer`	31
field	`n_head`	32
field	`n_kv_head`	33
field	`d_model`	34
field	`d_ff`	35
field	`dropout`	36
field	`tie_weights`	37
field	`rope_base`	38
field	`gradient_checkpointing`	39

Class `DataConfig`

```
class DataConfig
```

Data and batching configuration.

`micro_batch_size` is the batch size emitted by the loader. The trainer accumulates that many batches to reach `target_batch_size` when the latter is larger.

Method `__post_init__` {#api-speedtronic-config-DataConfig-post_init}

```
def __post_init__(self) -> None
```

Method `accumulation_steps`

```
def accumulation_steps(self) -> int
```

Class attributes and fields

Kind	Symbol	Line
field	`text_path`	71
field	`dataset`	72
field	`synthetic`	73
field	`num_tokens`	74
field	`vocab_size`	75
field	`block_size`	76
field	`micro_batch_size`	77
field	`target_batch_size`	78
field	`num_workers`	79
field	`prefetch_factor`	80
field	`pin_memory`	81
field	`shuffle`	82
field	`drop_last`	83
field	`seed`	84
field	`max_steps`	85

Class `OptimizerConfig`

```
class OptimizerConfig
```

Method `__post_init__` {#api-speedtronic-config-OptimizerConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`name`	112
field	`lr`	113
field	`betas`	114
field	`eps`	115
field	`weight_decay`	116
field	`fused`	117
field	`grad_clip`	118
field	`muon_plus`	119
field	`cautious`	120
field	`muon_momentum`	121
field	`muon_ns_steps`	122
field	`muon_norm_eps`	123

Class `SchedulerConfig`

```
class SchedulerConfig
```

Method `__post_init__` {#api-speedtronic-config-SchedulerConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`name`	151
field	`warmup_steps`	152
field	`max_steps`	153
field	`min_lr_ratio`	154

Class `PrecisionConfig`

```
class PrecisionConfig
```

Method `__post_init__` {#api-speedtronic-config-PrecisionConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`mode`	168
field	`dtype`	169

Class `CheckpointConfig`

```
class CheckpointConfig
```

Method `__post_init__` {#api-speedtronic-config-CheckpointConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`enabled`	183
field	`directory`	184
field	`every_steps`	185
field	`keep_last`	186
field	`resume`	187

Class `LoggingConfig`

```
class LoggingConfig
```

Method `__post_init__` {#api-speedtronic-config-LoggingConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`level`	198
field	`file`	199
field	`every_steps`	200
field	`json_file`	201

Class `ShapeValidationConfig`

```
class ShapeValidationConfig
```

Method `__post_init__` {#api-speedtronic-config-ShapeValidationConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`enabled`	211
field	`alignment`	212
field	`check_batch`	213
field	`check_sequence`	214
field	`check_model`	215
field	`check_vocab`	216
field	`warn_on_cpu`	217

Class `DistributedConfig`

```
class DistributedConfig
```

Method `__post_init__` {#api-speedtronic-config-DistributedConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`enabled`	247
field	`mode`	248
field	`role`	249
field	`node_id`	250
field	`collaborators`	251
field	`inner_steps`	252
field	`poll_interval`	253
field	`outer_lr`	254
field	`outer_momentum`	255
field	`repo_id`	256
field	`token`	257
field	`cache_dir`	258
field	`state_dir`	259
field	`retry_initial`	260
field	`retry_max`	261
field	`retry_attempts`	262
field	`reset_inner_optimizer`	263
field	`async_delta_upload`	264
field	`delta_upload_queue_size`	265
field	`delta_upload_overflow`	266
field	`delta_upload_shutdown_timeout`	267

Kind	Symbol	Line
field	`async_global_poll`	268

Class `RunConfig`

```
class RunConfig
```

Method `__post_init__` {#api-speedtronic-config-RunConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`name`	321
field	`seed`	322
field	`device`	323
field	`max_steps`	324
field	`output_dir`	325
field	`log_every`	326

Class `SpeedtronicConfig`

```
class SpeedtronicConfig
```

Method `__post_init__` {#api-speedtronic-config-SpeedtronicConfig-post_init}

```
def __post_init__(self) -> None
```

Method `accumulation_steps`

```
def accumulation_steps(self) -> int
```

Method `validate`

```
def validate(self) -> 'SpeedtronicConfig'
```

Validate cross-section constraints and return `self`.

Method `to_dict`

```
def to_dict(self, *, redact_secrets: bool=False) -> dict[str, Any]
```

Method `to_yaml`

```
def to_yaml(self, *, redact_secrets: bool=False) -> str
```

Method `save`

```
def save(self, path: str | os.PathLike[str]) -> Path
```

Method `from_dict`

```
def from_dict(cls, values: Mapping[str, Any] | None=None) -> 'SpeedtronicConfig'
```

Method `from_yaml`

```
def from_yaml(cls, path: str | os.PathLike[str]) -> 'SpeedtronicConfig'
```

Method `load`

```
def load(cls, source: str | os.PathLike[str] | Mapping[str, Any]) -> 'SpeedtronicConfig'
```

Class attributes and fields

Kind	Symbol	Line
field	`run`	337
field	`model`	338
field	`data`	339
field	`optimizer`	340
field	`scheduler`	341
field	`precision`	342
field	`checkpoint`	343
field	`logging`	344
field	`distributed`	345
field	`shape_validation`	346
field	`gradient_checkpointing`	347
field	`compile`	348
field	`ooo_backprop`	349
field	`ooo_streams`	350

Function `load_config`

```
def load_config(source: str | os.PathLike[str] | Mapping[str, Any]) -> SpeedtronicConfig
```

Function `load_yaml_config`

```
def load_yaml_config(path: str | os.PathLike[str]) -> SpeedtronicConfig
```

Class `_CompileSection`

```
class _CompileSection
```

Class attributes and fields

Kind	Symbol	Line
field	`enabled`	572

Function `_build_dataclass`

```
def _build_dataclass(cls: type[T], value: Any, path: str) -> T
```

Function `_reject_unknown`

```
def _reject_unknown(value: Mapping[str, Any], cls: type[Any], path: str) -> None
```

Function `_jsonable`

```
def _jsonable(value: Any) -> Any
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`Config`	556
constant/alias	`T`	567
constant/alias	`__all__`	615

`src/speedtronic/data.py`

Import path: `speedtronic.data`
Purpose: Dataset and dataloader utilities. The default data path is a small synthetic token stream so the project is usable immediately.
Lines: 269

Imported modules: `__future__.annotations`, `collections.abc.Callable`, `collections.abc.Iterable`, `collections.abc.Iterator`, `pathlib.Path`, `typing.Any`, `torch`, `torch.utils.data.DataLoader`, `torch.utils.data.Dataset`, `torch.utils.data.IterableDataset`, `torch.utils.data.get_worker_info`

Class `CharTokenizer`

```
class CharTokenizer
```

A deterministic byte-level tokenizer useful for examples and tests.

Method `__init__` {#api-speedtronic-data-CharTokenizer-init}

```
def __init__(self, vocab_size: int=256) -> None
```

Method `encode`

```
def encode(self, text: str) -> list[int]
```

Method `__call__` {#api-speedtronic-data-CharTokenizer-call}

```
def __call__(self, text: str) -> list[int]
```

Class `SyntheticTokenDataset`

```
class SyntheticTokenDataset(Dataset[dict[str, torch.Tensor]])
```

A reproducible random-token dataset for smoke tests and examples.

Method `__init__` {#api-speedtronic-data-SyntheticTokenDataset-init}

```
def __init__(self, num_samples: int=10000, block_size: int=128, vocab_size: int=512, seed: int=1234) ->
None
```

Method `__len__` {#api-speedtronic-data-SyntheticTokenDataset-len}

```
def __len__(self) -> int
```

Method `__getitem__` {#api-speedtronic-data-SyntheticTokenDataset-getitem}

```
def __getitem__(self, index: int) -> dict[str, torch.Tensor]
```

Class `TextFileTokenDataset`

```
class TextFileTokenDataset(IterableDataset[dict[str, torch.Tensor]])
```

Stream a text file in fixed-size token blocks.

`tokenizer` may be a callable or an object with `encode`. The file is read incrementally; only one block is materialized at a time.

Method `__init__` {#api-speedtronic-data-TextFileTokenDataset-init}

```
def __init__(self, path: str | Path, block_size: int, tokenizer: Callable[[str], Any] | Any | None=None,
vocab_size: int=256) -> None
```

Method `_encode`

```
def _encode(self, text: str) -> list[int]
```

Method `__iter__` {#api-speedtronic-data-TextFileTokenDataset-iter}

```
def __iter__(self) -> Iterator[dict[str, torch.Tensor]]
```

Function `collate_causal`

```
def collate_causal(batch: list[dict[str, torch.Tensor]]) -> dict[str, torch.Tensor]
```

Function `collate_batch`

```
def collate_batch(batch: list[Any]) -> Any
```

Collate causal dictionaries or homogeneous tuple/list datasets.

Function `_cycle`

```
def _cycle(loader: Iterable[Any]) -> Iterator[Any]
```

Function `infinite_batches`

```
def infinite_batches(loader: Iterable[Any]) -> Iterator[Any]
```

Cycle a finite loader, making max-step runs independent of data size.

Function `build_dataloader`

```
def build_dataloader(config: Any, *, dataset: Dataset | IterableDataset | None=None, tokenizer: Any |
None=None, pin_memory_device: bool=False, vocab_size: int | None=None) -> DataLoader
```

Build a loader from a DataConfig.

`dataset` is the primary extension point for user-provided datasets. A text path takes precedence over the synthetic fallback unless a dataset is explicitly supplied.

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	261

`src/speedtronic/distributed/__init__.py`

Import path: `speedtronic.distributed`
Purpose: DumbDiLoCo public API..
Lines: 31

Imported modules: `.diloco.DeltaUploadJob`, `.diloco.DumbDiLoCo`, `.diloco.DumbDiLoCoCoordinator`, `.diloco.SyncResult`, `.hub.GlobalMetadata`, `.hub.HubClient`, `.hub.HubError`, `.hub.HubTransport`, `.hub.HubUnavailable`, `.outer.MasterOuterLoop`, `.outer.NesterovOuterOptimizer`, `.tensors.average_deltas`, `.tensors.compute_pseudo_gradient`, `.tensors.load_delta`, `.tensors.load_safetensors`, `.tensors.save_safetensors`

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	14

`src/speedtronic/distributed/diloco.py`

Import path: `speedtronic.distributed.diloco`
Purpose: DumbDiLoCo coordinator integrated with the ordinary training loop..
Lines: 723

Imported modules: `__future__.annotations`, `logging`, `os`, `queue`, `threading`, `time`, `dataclasses.dataclass`, `pathlib.Path`, `typing.Any`, `torch`, `.hub.GlobalMetadata`, `.hub.HubClient`, `.outer.MasterOuterLoop`, `.tensors.compute_pseudo_gradient`, `.tensors.cpu_state_dict`, `.tensors.load_safetensors`

Class `SyncResult`

```
class SyncResult
```

Class attributes and fields

Kind	Symbol	Line
field	`pushed`	25
field	`loaded_global`	26
field	`outer_step`	27

Class `DeltaUploadJob`

```
class DeltaUploadJob
```

Immutable CPU snapshot dispatched to the asynchronous uploader.

Class attributes and fields

Kind	Symbol	Line
field	`step`	34
field	`base_outer_step`	35
field	`generation`	36
field	`delta`	37
field	`target_state`	38

Class `GlobalCandidate`

```
class GlobalCandidate
```

Class attributes and fields

Kind	Symbol	Line
field	`outer_step`	43
field	`state`	44

Class `DumbDiLoCoCoordinator`

```
class DumbDiLoCoCoordinator
```

Coordinate local inner loops and a Hub-backed outer loop.

The trainer calls :meth:`after_optimizer_step` after each local optimizer step. At `inner_steps` boundaries the coordinator computes and uploads a pseudo-gradient; at other steps it performs a cheap time-based poll for a newer global version. The master additionally owns a background :class:`MasterOuterLoop` thread.

Method `__init__` {#api-speedtronic-distributed-diloco-DumbDiLoCoCoordinator-init}

```
def __init__(self, config: Any, model: torch.nn.Module, *, hub: HubClient | None=None, state_dir: str | Path | None=None, logger: logging.Logger | None=None) -> None
```

Method `local_step`

```
def local_step(self) -> int
```

Method `last_global_step`

```
def last_global_step(self) -> int
```

Method `outer_step`

```
def outer_step(self) -> int
```

Method `_emit_event`

```
def _emit_event(self, event: str, payload: dict[str, Any]) -> None
```

Method `_start_io_workers`

```
def _start_io_workers(self) -> None
```

Method `start`

```
def start(self) -> None
```

Method `_start_master`

```
def _start_master(self) -> None
```

Method `_start_worker`

```
def _start_worker(self) -> None
```

Method `_load_global_state`

```
def _load_global_state(self, metadata: GlobalMetadata) -> dict[str, torch.Tensor]
```

Method `_refresh_from_hub`

```
def _refresh_from_hub(self, metadata: GlobalMetadata) -> bool
```

Method `_install_state`

```
def _install_state(self, state: dict[str, torch.Tensor]) -> None
```

Method `_on_outer_update`

```
def _on_outer_update(self, state: dict[str, torch.Tensor], outer_step: int) -> None
```

Method `_refresh_master_snapshot`

```
def _refresh_master_snapshot(self) -> bool
```

Method `_install_async_candidate`

```
def _install_async_candidate(self) -> bool
```

Method `_fetch_global_candidate`

```
def _fetch_global_candidate(self) -> GlobalCandidate \| None
```

Method `_schedule_async_poll`

```
def _schedule_async_poll(self, *, force: bool) -> bool
```

Method `_poll_worker`

```
def _poll_worker(self) -> None
```

Method `_consume_async_poll`

```
def _consume_async_poll(self) -> None
```

Method `_maybe_poll_global`

```
def _maybe_poll_global(self, *, force: bool=False) -> bool
```

Method `_upload_delta_sync`

```
def _upload_delta_sync(self, step: int) -> bool
```

Method `_dispatch_delta`

```
def _dispatch_delta(self, step: int) -> bool
```

Method `_upload_worker`

```
def _upload_worker(self) -> None
```

Method `_upload_delta`

```
def _upload_delta(self, step: int) -> bool
```

Method `after_optimizer_step`

```
def after_optimizer_step(self, model: torch.nn.Module, step: int) -> bool
```

Handle a local step; return whether the inner optimizer is reset.

Method `state_dict`

```
def state_dict(self) -> dict[str, Any]
```

Method `_restore_pending_upload`

```
def _restore_pending_upload(self, pending: dict[str, Any] | None) -> None
```

Method `load_state_dict`

```
def load_state_dict(self, state: dict[str, Any]) -> None
```

Method `prepare_state`

```
def prepare_state(self, state: dict[str, Any]) -> None
```

Stage resume state before startup performs any remote refresh.

Method `_apply_prepared_state`

```
def _apply_prepared_state(self) -> None
```

Method `stop`

```
def stop(self) -> None
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`LOGGER`	20
constant/alias	`DumbDiLoCo`	720
constant/alias	`__all__`	723

`src/speedtronic/distributed/hub.py`

Import path: `speedtronic.distributed.hub`
Purpose: Hugging Face Hub transport used as the DumbDiLoCo synchronization bus..
Lines: 329

Imported modules: `__future__.annotations`, `json`, `os`, `shutil`, `tempfile`, `time`, `dataclasses.dataclass`, `pathlib.Path`, `typing.Any`, `typing.Callable`, `.tensors.save_safetensors`

Class `HubError`

```
class HubError(RuntimeError)
```

Class `HubUnavailable`

```
class HubUnavailable(HubError)
```

Class `GlobalMetadata`

```
class GlobalMetadata
```

Method `from_dict`

```
def from_dict(cls, value: dict[str, Any]) -> 'GlobalMetadata'
```

Method `to_dict`

```
def to_dict(self) -> dict[str, Any]
```

Class attributes and fields

Kind	Symbol	Line
field	`outer_step`	27
field	`updated_at`	28
field	`model_file`	29

Class `HubClient`

```
class HubClient
```

A small retrying wrapper around `huggingface_hub`.

The wrapper intentionally exposes only repository-file operations. It is easy to replace with a fake object in tests and keeps the distributed code independent of Hub SDK version details.

Method `__init__` {#api-speedtronic-distributed-hub-HubClient-init}

```
def __init__(self, repo_id: str, *, token: str | None=None, cache_dir: str | os.PathLike[str] |
None=None, api: Any | None=None, retry_initial: float=1.0, retry_max: float=60.0, retry_attempts: int=6,
sleeper: Callable[[float], None]=time.sleep) -> None
```

Method `api`

```
def api(self) -> Any
```

Method `_retry`

```
def _retry(self, operation: str, callback: Callable[[], Any]) -> Any
```

Method `create_repo`

```
def create_repo(self, *, private: bool=True) -> Any
```

Method `add_collaborator`

```
def add_collaborator(self, username: str, permission: str='write') -> Any
```

Method `list_files`

```
def list_files(self) -> list[str]
```

Method `file_exists`

```
def file_exists(self, remote_path: str) -> bool
```

Method `upload`

```
def upload(self, local_path: str | os.PathLike[str], remote_path: str) -> Any
```

Method `download`

```
def download(self, remote_path: str, local_path: str | os.PathLike[str] | None=None) -> Path
```

Method `read_json`

```
def read_json(self, remote_path: str, *, default: Any=None) -> Any
```

Method `write_json`

```
def write_json(self, remote_path: str, value: Any) -> None
```

Method `global_metadata`

```
def global_metadata(self) -> GlobalMetadata \| None
```

Method `download_global`

```
def download_global(self, local_path: str | os.PathLike[str], metadata: GlobalMetadata | None=None) ->
Path
```

Method `publish_global`

```
def publish_global(self, state: dict[str, Any], *, outer_step: int, work_dir: str | os.PathLike[str] |
None=None, updated_at: str | None=None, metadata_extra: dict[str, str] | None=None) -> GlobalMetadata
```

Method `upload_delta`

```
def upload_delta(self, state: dict[str, Any], *, node_id: str, local_step: int, base_outer_step: int,
work_dir: str | os.PathLike[str] | None=None, metadata: dict[str, str] | None=None) -> str
```

Method `delta_paths`

```
def delta_paths(self) -> list[str]
```

Function `_utc_now`

```
def _utc_now() -> str
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`HubTransport`	320
constant/alias	`__all__`	329

`src/speedtronic/distributed/outer.py`

Import path: `speedtronic.distributed.outer`
Purpose: Outer-loop optimizer and master polling loop for DumbDiLoCo..
Lines: 361

Imported modules: `__future__.annotations`, `logging`, `threading`, `dataclasses.dataclass`, `dataclasses.field`, `datetime.datetime`, `datetime.timezone`, `pathlib.Path`, `typing.Any`, `typing.Callable`, `torch`, `..checkpoint.atomic_json_dump`, `..checkpoint.atomic_torch_save`, `.hub.HubClient`, `.tensors.average_deltas`, `.tensors.load_delta`, `.tensors.parse_delta_path`

Class `NesterovOuterOptimizer`

```
class NesterovOuterOptimizer
```

Nesterov momentum SGD over a floating-point model state.

Method `__init__` {#api-speedtronic-distributed-outer-NesterovOuterOptimizer-init}

```
def __init__(self, state: dict[str, torch.Tensor], lr: float, momentum: float=0.9) -> None
```

Method `step`

```
def step(self, state: dict[str, torch.Tensor], delta: dict[str, torch.Tensor]) -> None
```

Apply one outer update in place on CPU state tensors.

Method `state_dict`

```
def state_dict(self) -> dict[str, torch.Tensor]
```

Method `load_state_dict`

```
def load_state_dict(self, state: dict[str, torch.Tensor]) -> None
```

Class `OuterState`

```
class OuterState
```

Class attributes and fields

Kind	Symbol	Line
field	`global_state`	65
field	`outer_step`	66
field	`processed_deltas`	67
field	`momentum`	68
field	`last_error`	69

Class `MasterOuterLoop`

```
class MasterOuterLoop
```

Poll and aggregate deltas without coupling polling to inner training.

The loop owns a CPU copy of the global state and publishes a complete model plus metadata after every successful round. A private thread is used for the master role, so the training thread can continue its local loop.

Method `__init__` {#api-speedtronic-distributed-outer-MasterOuterLoop-init}

```
def __init__(self, hub: HubClient, state: dict[str, torch.Tensor], *, node_state_dir: str | Path,
outer_lr: float=0.7, outer_momentum: float=0.9, poll_interval: float=60.0, on_global_update:
Callable[[dict[str, torch.Tensor], int], None] | None=None, logger: logging.Logger | None=None) -> None
```

Method `processed_filenames`

```
def processed_filenames(self) -> set[str]
```

Method `state_path`

```
def state_path(self) -> Path
```

Method `metadata_path`

```
def metadata_path(self) -> Path
```

Method `_load_local_state`

```
def _load_local_state(self) -> None
```

Method `_save_local_state`

```
def _save_local_state(self) -> None
```

Method `_delta_candidates`

```
def _delta_candidates(self) -> list[tuple[str, int, int, str]]
```

Method `_download_delta`

```
def _download_delta(self, path: str) -> tuple[dict[str, torch.Tensor], dict[str, str]]
```

Method `sync_once`

```
def sync_once(self) -> int
```

Run one outer round and return the resulting outer step.

Hub listing and downloads happen outside the state lock. Only the short in-memory commit and local-state write are serialized, so a training-thread checkpoint never waits for a network round trip.

Method `_emit_outer_event`

```
def _emit_outer_event(self, found: int, valid: int) -> None
```

Method `adopt_global_state`

```
def adopt_global_state(self, state: dict[str, torch.Tensor], outer_step: int, *, reset_momentum:
bool=False) -> None
```

Recover a newer remotely published global state after a crash.

Method `snapshot`

```
def snapshot(self) -> tuple[int, dict[str, torch.Tensor]]
```

Method `start`

```
def start(self) -> None
```

Method `_run`

```
def _run(self) -> None
```

Method `stop`

```
def stop(self) -> None
```

Method `state_dict`

```
def state_dict(self) -> dict[str, Any]
```

Method `load_state_dict`

```
def load_state_dict(self, state: dict[str, Any]) -> None
```

Function `_utc_now`

```
def _utc_now() -> str
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`LOGGER`	18
constant/alias	`__all__`	361

`src/speedtronic/distributed/tensors.py`

Import path: `speedtronic.distributed.tensors`
Purpose: Safetensors helpers with defensive tensor normalization..
Lines: 148

Imported modules: `__future__.annotations`, `json`, `os`, `pathlib.Path`, `typing.Any`, `torch`

Function `cpu_tensor`

```
def cpu_tensor(value: torch.Tensor) -> torch.Tensor
```

Function `cpu_state_dict`

```
def cpu_state_dict(state: dict[str, torch.Tensor]) -> dict[str, torch.Tensor]
```

Function `save_safetensors`

```
def save_safetensors(state: dict[str, torch.Tensor], path: str | os.PathLike[str], *, metadata:
dict[str, str] | None=None) -> None
```

Function `load_safetensors`

```
def load_safetensors(path: str | os.PathLike[str]) -> dict[str, torch.Tensor]
```

Function `read_metadata`

```
def read_metadata(path: str | os.PathLike[str]) -> dict[str, str]
```

Function `floating_state`

```
def floating_state(state: dict[str, torch.Tensor]) -> dict[str, torch.Tensor]
```

Function `compute_pseudo_gradient`

```
def compute_pseudo_gradient(baseline: dict[str, torch.Tensor], current: dict[str, torch.Tensor]) ->
dict[str, torch.Tensor]
```

Compute `baseline - current` for floating-point state tensors.

Function `average_deltas`

```
def average_deltas(deltas: list[dict[str, torch.Tensor]]) -> dict[str, torch.Tensor]
```

Function `parse_delta_path`

```
def parse_delta_path(path: str) -> tuple[str, int] \| None
```

Function `load_delta`

```
def load_delta(path: str | os.PathLike[str]) -> tuple[dict[str, torch.Tensor], dict[str, str]]
```

Function `metadata_json`

```
def metadata_json(metadata: dict[str, str]) -> dict[str, Any]
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	136

`src/speedtronic/hooks.py`

Import path: `speedtronic.hooks`
Purpose: Optional hook and callback helpers..
Lines: 17

Imported modules: `__future__.annotations`, `typing.Any`, `typing.Callable`

Function `on_event`

```
def on_event(callback: Callable[[str, dict[str, Any]], None])
```

Return a callback wrapper suitable for `MetricLogger(hooks=...)`.

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	17

`src/speedtronic/integrations.py`

Import path: `speedtronic.integrations`
Purpose: Optional logging integrations loaded only when explicitly requested..
Lines: 38

Imported modules: `__future__.annotations`, `typing.Any`

Class `WandbHook`

```
class WandbHook
```

Method `__init__` {#api-speedtronic-integrations-WandbHook-init}

```
def __init__(self, project: str | None=None, **settings: Any) -> None
```

Method `on_event`

```
def on_event(self, event: str, payload: dict[str, Any]) -> None
```

Class `TensorboardHook`

```
class TensorboardHook
```

Method `__init__` {#api-speedtronic-integrations-TensorboardHook-init}

```
def __init__(self, log_dir: str='runs') -> None
```

Method `on_event`

```
def on_event(self, event: str, payload: dict[str, Any]) -> None
```

Method `close`

```
def close(self) -> None
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	38

`src/speedtronic/model.py`

Import path: `speedtronic.model`
Purpose: Reference decoder-only transformer used by the examples and smoke tests..
Lines: 345

Imported modules: `__future__.annotations`, `dataclasses.dataclass`, `typing.Any`, `torch`, `torch.nn`, `torch.nn.functional`, `torch.utils.checkpoint.checkpoint`, `.registry.register_model`

Class `GPTConfig`

```
class GPTConfig
```

Method `__post_init__` {#api-speedtronic-model-GPTConfig-post_init}

```
def __post_init__(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`vocab_size`	18
field	`block_size`	19
field	`n_layer`	20
field	`n_head`	21
field	`n_kv_head`	22
field	`d_model`	23
field	`d_ff`	24
field	`dropout`	25
field	`tie_weights`	26
field	`rope_base`	27

Class `RMSNorm`

```
class RMSNorm(nn.Module)
```

Method `__init__` {#api-speedtronic-model-RMSNorm-init}

```
def __init__(self, dim: int, eps: float=1e-05) -> None
```

Method `forward`

```
def forward(self, x: torch.Tensor) -> torch.Tensor
```

Function `_rotate_half`

```
def _rotate_half(x: torch.Tensor) -> torch.Tensor
```

Function `apply_rope`

```
def apply_rope(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor
```

Apply rotary embeddings to `x` with shape `(B, H, T, D)`.

Class `RotaryEmbedding`

```
class RotaryEmbedding(nn.Module)
```

Method `__init__` {#api-speedtronic-model-RotaryEmbedding-init}

```
def __init__(self, head_dim: int, base: float=10000.0) -> None
```

Method `forward`

```
def forward(self, seq_len: int, device: torch.device, dtype: torch.dtype) -> tuple[torch.Tensor,
torch.Tensor]
```

Class `CausalSelfAttention`

```
class CausalSelfAttention(nn.Module)
```

Method `__init__` {#api-speedtronic-model-CausalSelfAttention-init}

```
def __init__(self, config: GPTConfig) -> None
```

Method `forward`

```
def forward(self, x: torch.Tensor, rope: RotaryEmbedding, attention_mask: torch.Tensor | None=None) ->
torch.Tensor
```

Class `SwiGLU`

```
class SwiGLU(nn.Module)
```

Method `__init__` {#api-speedtronic-model-SwiGLU-init}

```
def __init__(self, config: GPTConfig) -> None
```

Method `forward`

```
def forward(self, x: torch.Tensor) -> torch.Tensor
```

Class `TransformerBlock`

```
class TransformerBlock(nn.Module)
```

Method `__init__` {#api-speedtronic-model-TransformerBlock-init}

```
def __init__(self, config: GPTConfig) -> None
```

Method `forward`

```
def forward(self, x: torch.Tensor, rope: RotaryEmbedding, attention_mask: torch.Tensor | None=None) ->
torch.Tensor
```

Method `_forward`

```
def _forward(self, x: torch.Tensor, rope: RotaryEmbedding, attention_mask: torch.Tensor | None=None) ->
torch.Tensor
```

Class `ReferenceTransformer`

```
class ReferenceTransformer(nn.Module)
```

A compact GPT-style model with RoPE, GQA, SwiGLU, and weight tying.

Method `__init__` {#api-speedtronic-model-ReferenceTransformer-init}

```
def __init__(self, **kwargs: Any) -> None
```

Method `_init_weights`

```
def _init_weights(module: nn.Module) -> None
```

Method `set_gradient_checkpointing`

```
def set_gradient_checkpointing(self, enabled: bool=True) -> None
```

Method `forward`

```
def forward(self, input_ids: torch.Tensor, labels: torch.Tensor | None=None, attention_mask:
torch.Tensor | None=None, **_: Any) -> dict[str, torch.Tensor]
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`GPT`	324
constant/alias	`Transformer`	325
constant/alias	`ReferenceModel`	326
constant/alias	`GPTModel`	327
constant/alias	`TransformerConfig`	328
constant/alias	`__all__`	331

`src/speedtronic/module_utils.py`

Import path: `speedtronic.module_utils`
Purpose: Small public helpers for user modules..
Lines: 24

Imported modules: `__future__.annotations`, `typing.Any`

Function `set_gradient_checkpointing`

```
def set_gradient_checkpointing(module: Any, enabled: bool=True) -> Any
```

Enable a module's explicit gradient-checkpointing convention.

Speedtronic intentionally does not inspect transformer internals. A custom module opts in by exposing `set_gradient_checkpointing(enabled)`.

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	24

`src/speedtronic/optimizers.py`

Import path: `speedtronic.optimizers`
Purpose: v2 optimizers: pure-PyTorch Muon, Muon+, cautious wrapping, and hybrid routing..
Lines: 552

Imported modules: `__future__.annotations`, `dataclasses.dataclass`, `typing.Any`, `typing.Iterable`, `typing.Mapping`, `typing.Sequence`, `torch`, `torch.nn`, `torch.optim.Optimizer`

Class `ParameterRouting`

```
class ParameterRouting
```

Deterministic parameter ownership for the hybrid optimizer.

Class attributes and fields

Kind	Symbol	Line
field	`muon`	17
field	`adamw`	18
field	`skipped`	19

Function `_work_dtype`

```
def _work_dtype(value: torch.Tensor) -> torch.dtype
```

Function `newton_schulz`

```
def newton_schulz(matrix: torch.Tensor, steps: int=5, eps: float=1e-07) -> torch.Tensor
```

Approximate the orthogonal polar factor with a quintic iteration.

The implementation intentionally uses only PyTorch matrix operations. It does not require a custom CUDA extension, SVD, or a device-specific package.

Function `post_polar_normalize`

```
def post_polar_normalize(update: torch.Tensor, *, mode: str='row_col', eps: float=1e-08) -> torch.Tensor
```

Apply Muon+'s inexpensive post-orthogonalization normalization.

Function `_parameter_owners`

```
def _parameter_owners(model: nn.Module) -> Mapping[int, tuple[str, nn.Module, str]]
```

Function `_is_embedding_or_head`

```
def _is_embedding_or_head(name: str, module: nn.Module) -> bool
```

Function `route_parameters`

```
def route_parameters(model: nn.Module) -> ParameterRouting
```

Route trainable parameters to Muon or AdamW by role and dimensionality.

Hidden `nn.Linear` matrices are eligible for Muon. Embeddings, heads, normalization parameters, biases, non-2D tensors, and frozen parameters are sent to AdamW or omitted. A module can opt into a route with `_speedtronic_optimizer_role = "muon"` or `"adamw"`.

Class `Muon`

```
class Muon(Optimizer)
```

A pure-PyTorch Muon optimizer for routed 2-D parameter groups.

Method `__init__` {#api-speedtronic-optimizers-Muon-init}

```
def __init__(self, params: Iterable[torch.Tensor], lr: float=0.0003, momentum: float=0.95, nesterov: bool=True, ns_steps: int=5, weight_decay: float=0.0, eps: float=1e-07, norm_eps: float=1e-08, muon_plus: bool=False) -> None
```

Method `step`

```
def step(self, closure: Any | None=None) -> Any
```

Class `HybridOptimizer`

```
class HybridOptimizer(Optimizer)
```

One optimizer facade combining Muon matrices and AdamW parameters.

Method `__init__` {#api-speedtronic-optimizers-HybridOptimizer-init}

```
def __init__(self, muon_params: Sequence[torch.Tensor], adamw_params: Sequence[torch.Tensor], *,
muon_lr: float, adamw_lr: float, betas: tuple[float, float], eps: float, weight_decay: float, fused:
bool=False, muon_momentum: float=0.95, muon_ns_steps: int=5, muon_norm_eps: float=1e-08, muon_plus:
bool=False) -> None
```

Method `_bind_children`

```
def _bind_children(self) -> None
```

Method `_sync_children`

```
def _sync_children(self) -> None
```

Method `step`

```
def step(self, closure: Any | None=None) -> Any
```

Method `load_state_dict`

```
def load_state_dict(self, state_dict: dict[str, Any]) -> None
```

Class `CautiousOptimizer`

```
class CautiousOptimizer(Optimizer)
```

Composable cautious wrapper around a Speedtronic/native optimizer.

The wrapper snapshots parameters, lets the base optimizer calculate its update, and masks entries whose observed update does not align with the current gradient. It intentionally supports arbitrary base optimizers; native AdamW and Muon remain available for exact direction-specific integrations.

Method `__init__` {#api-speedtronic-optimizers-CautiousOptimizer-init}

```
def __init__(self, base: Optimizer) -> None
```

Method `step`

```
def step(self, closure: Any | None=None) -> Any
```

Method `zero_grad`

```
def zero_grad(self, set_to_none: bool=True) -> None
```

Method `state_dict`

```
def state_dict(self) -> dict[str, Any]
```

Method `load_state_dict`

```
def load_state_dict(self, state_dict: dict[str, Any]) -> None
```

Method `add_param_group`

```
def add_param_group(self, param_group: dict[str, Any]) -> None
```

Function `build_v2_optimizer`

```
def build_v2_optimizer(model: nn.Module, config: Any, device: torch.device) -> Optimizer
```

Build AdamW, hybrid Muon/AdamW, and cautious variants from config.

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	543

`src/speedtronic/precision.py`

Import path: `speedtronic.precision`
Purpose: Hardware capability detection and mixed-precision helpers..
Lines: 147

Imported modules: `__future__.annotations`, `contextlib.nullcontext`, `dataclasses.dataclass`, `typing.Any`

Class `PrecisionPlan`

```
class PrecisionPlan
```

Method `is_mixed_precision`

```
def is_mixed_precision(self) -> bool
```

Class attributes and fields

Kind	Symbol	Line
field	`mode`	12
field	`dtype`	13
field	`use_scaler`	14
field	`autocast_enabled`	15
field	`device_type`	16

Function `resolve_device`

```
def resolve_device(requested: str | Any='auto') -> Any
```

Function `resolve_precision`

```
def resolve_precision(config: Any, device: Any) -> PrecisionPlan
```

Resolve a precision config against the actual device.

The explicit mode wins, except that an explicit unsupported mixed mode falls back to a safe mode rather than making a run unusable.

Function `autocast_context`

```
def autocast_context(plan: PrecisionPlan, device: Any)
```

Function `make_grad_scaler`

```
def make_grad_scaler(plan: PrecisionPlan)
```

Function `supports_fused_adamw`

```
def supports_fused_adamw(device: Any) -> bool
```

Function `parameter_count`

```
def parameter_count(model: Any) -> int
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	140

`src/speedtronic/profiling.py`

Import path: `speedtronic.profiling`
Purpose: Lightweight stdout/file metrics and pluggable training hooks..
Lines: 147

Imported modules: `__future__.annotations`, `json`, `logging`, `sys`, `threading`, `time`, `dataclasses.dataclass`, `pathlib.Path`, `typing.Any`, `typing.Callable`, `typing.Protocol`, `typing.TextIO`

Class `TrainingHook`

```
class TrainingHook(Protocol)
```

Method `on_event`

```
def on_event(self, event: str, payload: dict[str, Any]) -> None
```

Function `memory_usage_bytes`

```
def memory_usage_bytes() -> int \| None
```

Return allocated accelerator memory when the backend exposes it.

Class `MetricLogger`

```
class MetricLogger
```

Method `__post_init__` {#api-speedtronic-profiling-MetricLogger-post_init}

```
def __post_init__(self) -> None
```

Method `emit`

```
def emit(self, event: str, payload: Metric) -> None
```

Method `_emit_locked`

```
def _emit_locked(self, event: str, payload: Metric) -> None
```

Method `log`

```
def log(self, level: int, message: str) -> None
```

Method `debug`

```
def debug(self, message: str, *args: Any) -> None
```

Method `info`

```
def info(self, message: str, *args: Any) -> None
```

Method `warning`

```
def warning(self, message: str, *args: Any) -> None
```

Method `error`

```
def error(self, message: str, *args: Any) -> None
```

Method `close`

```
def close(self) -> None
```

Class attributes and fields

Kind	Symbol	Line
field	`level`	40
field	`file`	41
field	`json_file`	42
field	`every_steps`	43
field	`stream`	44
field	`hooks`	45

Function `_format_metrics`

```
def _format_metrics(metrics: Metric) -> str
```

Class `CallbackList`

```
class CallbackList
```

Fan-out adapter for optional W&B/TensorBoard-style callbacks.

Method `__init__` {#api-speedtronic-profiling-CallbackList-init}

```
def __init__(self, callbacks: list[Any] | None=None) -> None
```

Method `on_event`

```
def on_event(self, event: str, payload: dict[str, Any]) -> None
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`Metric`	14
constant/alias	`Hook`	15
constant/alias	`__all__`	147

`src/speedtronic/registry.py`

Import path: `speedtronic.registry`
Purpose: Model registry used to keep the training engine architecture-neutral..
Lines: 75

Imported modules: `__future__.annotations`, `collections.abc.Callable`, `typing.Any`

Class `ModelRegistry`

```
class ModelRegistry
```

Method `__init__` {#api-speedtronic-registry-ModelRegistry-init}

```
def __init__(self) -> None
```

Method `register`

```
def register(self, name: str, factory: ModelFactory | None=None)
```

Register a factory, usable as a decorator or a normal function.

Method `get`

```
def get(self, name: str) -> ModelFactory
```

Method `names`

```
def names(self) -> tuple[str, ...]
```

Method `build`

```
def build(self, name: str, **kwargs: Any) -> Any
```

Class attributes and fields

Kind	Symbol	Line
class attribute	`_builtin_names`	12

Function `register_model`

```
def register_model(name: str, factory: ModelFactory | None=None)
```

Function `build_model`

```
def build_model(config: Any, **overrides: Any) -> Any
```

Build a model from a :class:`~speedtronic.config.ModelConfig`.

Extra keyword arguments are useful for custom models and are intentionally passed through without interpretation.

Module constants and aliases

Kind	Symbol	Line
constant/alias	`ModelFactory`	8
constant/alias	`registry`	44
constant/alias	`__all__`	75

`src/speedtronic/runtime.py`

Import path: `speedtronic.runtime`
Purpose: Composition helpers that turn one config into a runnable trainer..
Lines: 173

Imported modules: `__future__.annotations`, `math`, `os`, `random`, `pathlib.Path`, `typing.Any`, `torch`, `.checkpoint.CheckpointManager`, `.config.SpeedtronicConfig`, `.data.build_dataloader`, `.distributed.DumbDiLoCoCoordinator`, `.optimizers.build_v2_optimizer`, `.precision.resolve_device`, `.precision.resolve_precision`, `.profiling.MetricLogger`, `.registry.build_model`, `.shapes.ShapeReport`, `.shapes.validate_startup_shapes`, `.trainer.Trainer`, `.trainer.TrainResult`

Function `seed_everything`

```
def seed_everything(seed: int) -> None
```

Function `build_optimizer`

```
def build_optimizer(model: torch.nn.Module, config: Any, device: Any) -> torch.optim.Optimizer
```

Build the configured AdamW, hybrid Muon, or cautious optimizer.

Function `build_scheduler`

```
def build_scheduler(optimizer: torch.optim.Optimizer, config: SpeedtronicConfig) ->
torch.optim.lr_scheduler.LambdaLR
```

Function `build_logger`

```
def build_logger(config: SpeedtronicConfig) -> MetricLogger
```

Function `build_runtime`

```
def build_runtime(config: SpeedtronicConfig | dict[str, Any], *, resume: bool=False, device: str |
torch.device | None=None, max_steps: int | None=None) -> tuple[Trainer, CheckpointManager]
```

Build all v1 components from one configuration object.

Function `train_from_config`

```
def train_from_config(config: SpeedtronicConfig | dict[str, Any], *, resume: bool=False, device: str |
torch.device | None=None, max_steps: int | None=None) -> TrainResult
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`run`	160
constant/alias	`train`	161
constant/alias	`__all__`	164

`src/speedtronic/scheduling.py`

Import path: `speedtronic.scheduling`
Purpose: Dependency-aware forward/backward stream scheduling for v2..
Lines: 220

Imported modules: `__future__.annotations`, `dataclasses.dataclass`, `typing.Any`, `typing.Iterable`, `torch`, `torch.nn`

Class `StageInfo`

```
class StageInfo
```

Class attributes and fields

Kind	Symbol	Line
field	`name`	14
field	`stream_index`	15

Function `_iter_tensors`

```
def _iter_tensors(value: Any) -> Iterable[torch.Tensor]
```

Function `_expand_stage_modules`

```
def _expand_stage_modules(model: nn.Module) -> list[tuple[str, nn.Module]]
```

Find disjoint, meaningful stage roots without nesting hooks.

Class `StageStreamScheduler`

```
class StageStreamScheduler
```

Run disjoint module stages on CUDA streams for autograd's DAG engine.

PyTorch's autograd engine already performs dependency-aware out-of-order node execution. The engine still routes a node to the forward stream that created it, so this scheduler assigns disjoint forward stages to multiple streams. Cross-stream inputs wait on their producer and `record_stream` protects allocator reuse. CPU and MPS intentionally fall back to the standard sequential path.

Method `__init__` {#api-speedtronic-scheduling-StageStreamScheduler-init}

```
def __init__(self, model: nn.Module, device: torch.device | str, *, num_streams: int=4, logger: Any |
None=None) -> None
```

Method `_install_hooks`

```
def _install_hooks(self, modules: list[tuple[str, nn.Module]]) -> None
```

Method `remove_hooks`

```
def remove_hooks(self) -> None
```

Method `_pre_hook`

```
def _pre_hook(self, module: nn.Module, args: tuple[Any, ...], kwargs: dict[str, Any])
```

Method `_post_hook`

```
def _post_hook(self, module: nn.Module, args: tuple[Any, ...], output: Any)
```

Method `begin`

```
def begin(self) -> None
```

Method `finish`

```
def finish(self) -> None
```

Method `dispose`

```
def dispose(self) -> None
```

Method `as_dict`

```
def as_dict(self) -> dict[str, Any]
```

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	220

`src/speedtronic/shapes.py`

Import path: `speedtronic.shapes`
Purpose: Startup shape-efficiency warnings for configured batches and model GEMMs..
Lines: 244

Imported modules: `__future__.annotations`, `dataclasses.asdict`, `dataclasses.dataclass`, `dataclasses.field`, `typing.Any`, `typing.Mapping`, `torch`, `torch.nn`, `.precision.PrecisionPlan`

Class `ShapeProfile`

```
class ShapeProfile
```

Class attributes and fields

Kind	Symbol	Line
field	`device_type`	16
field	`precision`	17
field	`alignment`	18
field	`source`	19

Class `ShapeWarning`

```
class ShapeWarning
```

Class attributes and fields

Kind	Symbol	Line
field	`code`	24
field	`field`	25
field	`value`	26
field	`suggested`	27
field	`alignment`	28
field	`reason`	29

Class `ShapeReport`

```
class ShapeReport
```

Method `warning_count`

```
def warning_count(self) -> int
```

Method `as_dict`

```
def as_dict(self) -> dict[str, Any]
```

Class attributes and fields

Kind	Symbol	Line
field	`profile`	34
field	`warnings`	35

Function `_suggest`

```
def _suggest(value: int, alignment: int) -> int
```

Function `resolve_shape_profile`

```
def resolve_shape_profile(device: torch.device | str, precision: PrecisionPlan, *, requested_alignment:
int | str | None='auto') -> ShapeProfile
```

Function `_append_warning`

```
def _append_warning(warnings: list[ShapeWarning], seen: set[tuple[str, int, int]], *, code: str, name:
str, value: int, alignment: int, reason: str, suppress_suggestion: bool=False) -> None
```

Function `_module_metadata`

```
def _module_metadata(model: nn.Module) -> list[tuple[str, int]]
```

Function `validate_startup_shapes`

```
def validate_startup_shapes(config: Any, model: nn.Module, device: torch.device | str, precision:
PrecisionPlan, *, logger: Any | None=None) -> ShapeReport
```

Inspect effective shapes and return non-fatal efficiency warnings.

Module constants and aliases

Kind	Symbol	Line
constant/alias	`__all__`	238

`src/speedtronic/trainer.py`

Import path: `speedtronic.trainer`
Purpose: Architecture-neutral training loop..
Lines: 528

Imported modules: `__future__.annotations`, `time`, `dataclasses.dataclass`, `typing.Any`, `typing.Iterable`, `typing.Protocol`, `torch`, `torch.nn.functional`, `.checkpoint.CheckpointManager`, `.checkpoint.capture_rng_state`, `.checkpoint.restore_rng_state`, `.data.infinite_batches`, `.precision.PrecisionPlan`, `.precision.autocast_context`, `.precision.make_grad_scaler`, `.precision.resolve_precision`, `.profiling.MetricLogger`, `.scheduling.StageStreamScheduler`, `.shapes.ShapeReport`, `.shapes.validate_startup_shapes`

Class `SyncCoordinator`

```
class SyncCoordinator(Protocol)
```

Method `start`

```
def start(self) -> None
```

Method `after_optimizer_step`

```
def after_optimizer_step(self, model: Any, step: int) -> bool \| None
```

Method `stop`

```
def stop(self) -> None
```

Method `state_dict`

```
def state_dict(self) -> dict[str, Any] \| None
```

Method `load_state_dict`

```
def load_state_dict(self, state: dict[str, Any]) -> None
```

Class `TrainResult`

```
class TrainResult
```

Class attributes and fields

Kind	Symbol	Line
field	`steps`	34
field	`samples`	35
field	`tokens`	36
field	`final_loss`	37
field	`elapsed_s`	38
field	`metrics`	39

Class `Trainer`

```
class Trainer
```

Train a user-provided module with optional local or DumbDiLoCo sync.

The model receives a batch dictionary when the loader emits dictionaries; otherwise the first two tuple elements are passed as `inputs, labels`. A model may return a loss directly, a mapping with `loss`, or a tuple whose first element is the loss.

Method `__init__` {#api-speedtronic-trainer-Trainer-init}

```
def __init__(self, model: torch.nn.Module, optimizer: torch.optim.Optimizer, dataloader:
Iterable[dict[str, torch.Tensor]], *, device: torch.device | str, config: Any | None=None, scheduler:
Any | None=None, precision: PrecisionPlan | None=None, logger: MetricLogger | None=None,
checkpoint_manager: CheckpointManager | None=None, coordinator: SyncCoordinator | None=None, max_steps:
int | None=None, start_step: int=0, shape_report: ShapeReport | None=None) -> None
```

Method `accumulation_steps`

```
def accumulation_steps(self) -> int
```

Method `_configure_features`

```
def _configure_features(self) -> None
```

Method `_setup_ooo_backprop`

```
def _setup_ooo_backprop(self) -> None
```

Method `_enable_compile`

```
def _enable_compile(self) -> None
```

Method `_move_batch`

```
def _move_batch(self, batch: Any) -> Any
```

Method `_batch_labels`

```
def _batch_labels(batch: Any) -> torch.Tensor \| None
```

Method `_loss_from_logits`

```
def _loss_from_logits(cls, output: Any, batch: Any) -> torch.Tensor \| None
```

Method `_forward`

```
def _forward(self, batch: Any) -> tuple[torch.Tensor, dict[str, Any]]
```

Method `_forward_with_compile_fallback`

```
def _forward_with_compile_fallback(self, batch: Any) -> tuple[torch.Tensor, dict[str, Any]]
```

Method `_batch_size_and_tokens`

```
def _batch_size_and_tokens(batch: Any) -> tuple[int, int]
```

Method `_optimizer_step`

```
def _optimizer_step(self, loss: torch.Tensor, final_microbatch: bool, loss_scale: float=1.0) -> float
```

Method `_reset_inner_optimizer_if_needed`

```
def _reset_inner_optimizer_if_needed(self) -> None
```

Method `_checkpoint`

```
def _checkpoint(self, *, force: bool=False) -> None
```

Method `resume`

```
def resume(self, state: dict[str, Any]) -> None
```

Method `fit`

```
def fit(self, max_steps: int | None=None) -> TrainResult
```

Method `_compiled_model`

```
def _compiled_model(self) -> torch.nn.Module
```

Class attributes and fields

Kind	Symbol	Line
class attribute	`train`	504

Function `_checkpoint_config`

```
def _checkpoint_config(config: Any) -> Any
```

Class `_AutoPrecision`

```
class _AutoPrecision
```

Class attributes and fields

Kind	Symbol	Line
class attribute	`mode`	520
class attribute	`dtype`	521

Module constants and aliases

Kind	Symbol	Line
constant/alias	`TrainingEngine`	524
constant/alias	`SpeedtronicTrainer`	525
constant/alias	`__all__`	528

Test modules

Every test function defined by the repository is listed here.

`tests/test_config.py`

- `test_config::test_config_yaml_aliases_and_accumulation`
- `test_config::test_config_round_trip`
- `test_config::test_unknown_keys_and_invalid_batch`
- `test_config::test_distributed_role_inference`
- `test_config::test_secret_can_be_redacted_for_serialization`

`tests/test_distributed.py`

- `test_distributed::test_pseudo_gradient_direction_and_average`
- `test_distributed::test_nesterov_outer_optimizer`
- `test_distributed::test_delta_path_parser`
- `test_distributed::test_tied_model_state_can_be_safetensors_serialized`

`tests/test_hub.py`

- `test_hub::test_hub_transport_round_trip`
- `test_hub::test_master_skips_corrupt_delta_and_persists_processed_set`

`tests/test_model.py`

- `test_model::test_reference_transformer_forward_and_loss`
- `test_model::test_reference_transformer_weight_tying_and_checkpoint_hook`
- `test_model::test_config_dimensions`

`tests/test_precision_data.py`

- `test_precision_data::test_cpu_precision_defaults_to_fp32`
- `test_precision_data::test_streaming_text_dataset_and_tuple_collate`

`tests/test_training.py`

- `test_training::test_trainer_runs_with_accumulation_and_checkpoint`
- `test_training::test_accumulation_scales_gradients_and_reports_mean_loss`
- `test_training::test_repeated_fit_restarts_coordinator_lifecycle`
- `test_training::test_runtime_builds_reference_model`

tests/test_v2_distributed.py

- test_v2_distributed::test_async_global_poll_installs_on_training_thread
- test_v2_distributed::test_coordinator_can_restart_after_stop
- test_v2_distributed::test_async_delta_dispatch_does_not_block_training
- test_v2_distributed::test_prepared_resume_restores_pending_upload
- test_v2_distributed::test_synchronous_delta_mode_remains_available
- test_v2_distributed::test_async_delta_failure_keeps_baseline_for_cumulative_retry

tests/test_v2_optimizers.py

- test_v2_optimizers::test_v2_config_accepts_scalar_optimizer_and_root_flags
- test_v2_optimizers::test_muon_plus_requires_muon
- test_v2_optimizers::test_v2_config_rejects_invalid_systems_values
- test_v2_optimizers::test_newton_schulz_is_pure_and_finite
- test_v2_optimizers::test_newton_schulz_handles_rectangular_matrices
- test_v2_optimizers::test_post_polar_normalization_produces_unit_rows_and_columns
- test_v2_optimizers::test_reference_parameter_routing_is_role_aware_and_tied_safe
- test_v2_optimizers::test_hybrid_optimizer_updates_both_branches_and_scheduler
- test_v2_optimizers::test_adamw_never_builds_an_empty_optimizer_for_all_linear_model
- test_v2_optimizers::test_non_linear_custom_matrix_defaults_to_adamw
- test_v2_optimizers::test_hybrid_optimizer_forwards_closure_once
- test_v2_optimizers::test_cautious_wrapper_is_composable_and_finite
- test_v2_optimizers::test_muon_rejects_non_matrix_parameters
- test_v2_optimizers::test_muon_sparse_gradient_is_rejected
- test_v2_optimizers::test_v2_muon_runtime_smoke

tests/test_v2_release.py

- test_v2_release::test_version_is_synchronized
- test_v2_release::test_cli_validate_redacts_configured_token
- test_v2_release::test_v2_config_serializes_all_new_sections

tests/test_v2_systems.py

- test_v2_systems::test_cpu_shape_auto_profile_is_quiet_but_explicit_alignment_warns
- test_v2_systems::test_shape_report_is_non_fatal_and_deduplicated
- test_v2_systems::test_explicit_cpu_profile_requires_opt_in_and_avoids_invalid_batch_suggestion
- test_v2_systems::test_shape_validation_accepts_standard_library_logger
- test_v2_systems::test_scheduler_default_follows_run_target

- `test_v2_systems::test_causal_inferred_loss_uses_already_shifted_labels`
- `test_v2_systems::test_out_of_order_scheduler_degrades_to_noop_on_cpu`
- `test_v2_systems::test_out_of_order_config_runs_without_changing_cpu_path`

Shipped configurations and examples

- `configs/diloco.yaml`
- `configs/smoke.yaml`
- `configs/v2_smoke.yaml`
- `examples/diloco_master.yaml`
- `examples/train_reference.py`

Test map

The repository defines the v1 suite plus v2 optimizer, systems, and asynchronous distributed tests. The generated source inventory is authoritative for the current count. The tables below map the v1 and v2 tests to their direct evidence.

Configuration tests

`tests/test_config.py`

Test	Direct assertion	Primary source
`test_config_yaml_aliases_and_accumulation`	Top-level run aliases, derived accumulation, compile mapping	`config.py`
`test_config_round_trip`	Redacted YAML save/load equality	`config.py`
`test_unknown_keys_and_invalid_batch`	Unknown root and invalid batch divisibility fail	`config.py`
`test_distributed_role_inference`	Enabled repo-bearing role becomes master	`config.py`
`test_secret_can_be_redacted_for_serialization`	Explicit redaction and redacted `save()`	`config.py`

Not directly tested:

- rare section value constraints;
- JSON and inline source loading;
- `hub` / `diloco` aliases;
- top-level versus nested run precedence;
- malformed YAML/JSON normalization.

Model tests

`tests/test_model.py`

Test	Direct assertion	Primary source
`test_reference_transformer_forward_and_loss`	Logits/loss shapes and backward gradients	`model.py`
`test_reference_transformer_weight_tying_and_checkpoint_hook`	Tied storage and block flag propagation	`model.py`
`test_config_dimensions`	`GPTConfig.d_ff` defaults to `4 * d_model`	`model.py`

Not directly tested:

- RoPE correctness;
- GQA equivalence/performance;
- SDPA mask behavior;
- causality and label alignment;
- SwiGLU shape;
- dropout/training mode;
- untied weights;
- invalid context lengths.

Precision and data tests

`tests/test_precision_data.py`

Test	Direct assertion	Primary source
`test_cpu_precision_defaults_to_fp32`	CPU auto FP32 and no scaler	`precision.py`
`test_cpu_precision_defaults_to_fp32` second assertion	Explicit CPU FP16 falls back to FP32	`precision.py`
`test_streaming_text_dataset_and_tuple_collate`	Text block shape and tuple stacking	`data.py`

Not directly tested:

- CUDA/MPS;
- CPU BF16;
- fused AdamW;
- scaler creation/restoration;
- DataLoader worker/prefetch/pin behavior;
- custom tokenizer;
- serialized datasets;
- empty streams.

Training tests

`tests/test_training.py`

Test	Direct assertion	Primary source
`test_trainer_runs_with_accumulation_and_checkpoint`	Three CPU steps, three metrics, checkpoint pointer	`trainer.py`, `checkpoint.py`
`test_accumulation_scales_gradients_and_reports_mean_loss`	Gradient scaling and mean loss value	`trainer.py`
`test_runtime_builds_reference_model`	Reference composition and one step	`runtime.py`, `model.py`

Not directly tested:

- checkpoint contents and full resume equivalence;
- DataLoader position;
- scheduler trajectory;
- clipping;
- compile fallback;
- every model output form;
- real CLI train error mapping;
- real coordinator lifecycle with Hub credentials.

Tensor and outer optimizer tests

`tests/test_distributed.py`

Test	Direct assertion	Primary source
`test_pseudo_gradient_direction_and_average`	Baseline-current direction and equal mean	`distributed/tensors.py`
`test_nesterov_outer_optimizer`	First update equals `1.9 × gradient` at momentum 0.9	`distributed/outer.py`
`test_delta_path_parser`	Valid node delta and rejected global path	`distributed/tensors.py`
`test_tied_model_state_can_be_safetensors_serialized`	Tied state serializes after clone	`distributed/tensors.py`

Not directly tested:

- key/shape mismatch errors;
- non-floating/complex behavior;
- delta metadata parsing;
- outer momentum persistence;
- large/NaN values.

Hub tests

`tests/test_hub.py`

Test	Direct assertion	Primary source
`test_hub_transport_round_trip`	Fake global and delta round trip plus write grant	`distributed/hub.py`
`test_master_skips_corrupt_delta_and_persists_processed_set`	Corrupt delta is skipped and no state file is created	`distributed/outer.py`

The second test does not actually verify successful processed-set persistence despite its name.

Not directly tested:

- retry/backoff;
- cached metadata fallback;
- real Hugging Face SDK versions;
- collaborator API compatibility;
- successful aggregation/publication;
- duplicate suppression;
- rollback/recovery;
- background thread lifecycle.

v2 test modules

File	Coverage
`tests/test_v2_optimizers.py`	Config, Newton–Schulz, Muon+, routing, hybrid/cautious stepping, sparse rejection
`tests/test_v2_systems.py`	Shape profiles, causal alignment, scheduler default, CPU OOO fallback
`tests/test_v2_distributed.py`	Non-blocking dispatch, async poll, prepared resume, queue skip, synchronous compatibility, failure baseline retention
`tests/test_v2_release.py`	Version sync, CLI redaction, v2 config serialization

CUDA-only stream parity and hardware benchmarks are not represented as CPU unit-test evidence.

Module evidence matrix

Module	Direct tests	Evidence level
`config.py`	`test_config.py`	Partial
`model.py`	`test_model.py`	Partial
`precision.py`	`test_precision_data.py`	CPU partial
`data.py`	`test_precision_data.py`, `test_training.py`	Partial
`trainer.py`	`test_training.py`	CPU partial
`runtime.py`	`test_training.py`	Minimal
`checkpoint.py`	`test_training.py`	Basic file/pointer only
`distributed/tensors.py`	`test_distributed.py`	Basic
`distributed/outer.py`	`test_distributed.py`, `test_hub.py`, `test_v2_distributed.py`	Basic plus lock-scope tests
`distributed/hub.py`	`test_hub.py`	Fake happy path
`distributed/diloco.py`	`test_v2_distributed.py`	Async dispatch/poll/resume coverage
`distributed/__init__.py`	Indirect imports	No direct test
`cli.py`	`test_v2_release.py`	Validation/redaction coverage
`registry.py`	None	No direct test
`profiling.py`	None	No direct test
`hooks.py`	None	No direct test
`integrations.py`	None	No direct test
`module_utils.py`	Indirect model hook only	No helper test
`__init__.py`	None	No export/lazy-load test

Module	Direct tests	Evidence level
`__main__.py`	None	No module-entry test

Test philosophy

Tests are intended to be CPU-only and use fake Hub APIs. They make no real network calls and require no credentials. Hardware-specific claims remain implementation-derived unless directly exercised.

The complete source-level inventory also includes every test function generated from the AST. See Generated Source Inventory.

Related: Testing, Source Coverage, and Security and Limitations.

Glossary

Absolute step

A global optimizer-step target rather than a count of additional updates. A trainer at step 900 targeting 1000 performs 100 updates.

Accumulation

Multiple microbatches whose gradients are summed and applied in one optimizer update. Speedtronic divides each microbatch loss by the accumulation count.

AdamW

The decoupled weight-decay optimizer used by the framework. Optional fused mode is attempted only on supported CUDA setups.

AMP

Automatic mixed precision. Speedtronic uses autocast for BF16/FP16 and a CUDA `GradScaler` for FP16.

Baseline

In DumbDiLoCo, the model state captured at the last successful delta upload or global installation. The next boundary delta is baseline minus current.

BF16

A floating-point format with a wider exponent range and lower precision than FP32. Commonly native on supported CUDA hardware.

Block size

The number of input positions in a causal model sample. The reference model rejects sequences longer than its configured block size.

Causal LM loss

Next-token cross-entropy computed from logits and shifted labels, usually ignoring padding index `-100`.

Checkpoint

A local pickle-backed state file containing model and training state, optionally including coordinator state.

Compile fallback

Speedtronic's behavior of disabling `torch.compile` and rerunning a failed compiled forward eagerly.

DataLoader position

The current epoch, sampler, batch, and worker iterator state. Speedtronic 2.0.0 does not checkpoint this state.

Distributed

The optional DumbDiLoCo mode. It does not use NCCL or `torch.distributed` collectives.

Dilation/outer step

See **Outer step**.

DumbDiLoCo

Speedtronic's Hub-transport, DiLoCo-style protocol: local steps, pseudo-gradient upload, mean aggregation, Nesterov outer update, and global-weight polling.

FP16

A half-precision floating-point format. On CUDA it uses `GradScaler`; CPU explicit FP16 falls back to FP32.

FP32

Standard 32-bit floating-point execution. It is the portable default on CPU/MPS and the fallback for several unsupported mixed modes.

GQA

Grouped-query attention. The reference model projects fewer K/V heads than query heads and repeats K/V groups to match query-head count.

Global model

The complete model `state_dict` published by the master at `global/latest.safetensors`.

Global step

In normal local training, the cumulative optimizer update count. In DumbDiLoCo Hub metadata, `outer_step` is the global model version; avoid using the same word for both in operational logs.

Gradient accumulation

See **Accumulation**.

Gradient checkpointing

Activation-memory optimization that recomputes selected forward work during backward. The model opts in through `set_gradient_checkpointing(enabled)`.

Hub

The Hugging Face Hub. Speedtronic uses a model repository as mutable file transport for global state and deltas.

Inner boundary

A local step divisible by `distributed.inner_steps`.

Inner loop/step

The local optimization loop. In Speedtronic's coordinator, an inner step is one local optimizer update after gradient accumulation.

LambdaLR

A PyTorch scheduler driven by a function of the optimizer-step count. Speedtronic builds warmup and cosine/constant factors.

Local step

One completed local AdamW optimizer update and scheduler step before the coordinator callback.

LR

Learning rate.

Microbatch

One loader batch consumed before an optimizer update. The Trainer forward/backward path runs once per microbatch.

MPS

Apple Metal Performance Shaders backend exposed by PyTorch. Speedtronic auto-selects it after CUDA and before CPU, but uses FP32 by default.

Nesterov momentum SGD

The outer optimizer update:

```
buffer = momentum * buffer + gradient
effective = gradient + momentum * buffer
state -= outer_lr * effective
```

Node ID

The distributed participant's unique path-safe identity, used for `nodes/<node_id>/...` and local state directories.

Outer loop

Master-side aggregation and global optimization over uploaded pseudo-gradients.

Outer optimizer

`NesterovOuterOptimizer`, which updates the CPU global state from the mean valid delta.

Outer round

One successful `MasterOuterLoop.sync_once()` aggregation and publication.

Outer step

The monotonic global model version stored in `global/step_count.json`. The name `outer_step` distinguishes it from local optimizer steps.

Parameter server

A centralized service hosting model/optimizer state. DumbDiLoCo does not use one; the Hub is file transport.

Persistent workers

DataLoader worker processes kept alive across epochs. Enabled automatically when `num_workers > 0`.

Pin memory

Page-locking CPU tensor storage to accelerate CUDA transfers. Configured explicitly or inferred for resolved CUDA devices.

Prefetch factor

Number of batches each DataLoader worker prepares in advance. Defaults to 2 when workers are enabled.

Pseudo-gradient

```
baseline weights - current local weights
```

It approximates a direction for the outer global update and is not an autograd gradient.

Registry

The process-global mapping from model name to factory. Built-ins are `reference_transformer` and `gpt`.

Resume

Loading model, optimizer, scheduler, scaler, counters, RNG, and optional coordinator state from a local checkpoint. It is not exact DataLoader replay.

RMSNorm

Root-mean-square layer normalization with a learned scale, implemented in the reference model.

RoPE

Rotary positional embedding. Query and key vectors are rotated by position-dependent angles before attention.

Safetensors

A tensor serialization format used for global models and deltas. It avoids pickle execution for tensor files but does not authenticate writers.

Scheduler horizon

`scheduler.max_steps`, the step count used to shape the cosine schedule. It can differ from the actual run target if not configured explicitly.

SDPA

PyTorch's `scaled_dot_product_attention`, used by the reference model for backend-selected attention kernels.

Staleness

How old a worker's baseline/global version is. DumbDiLoCo records `base_outer_step` but does not use it to reject or weight stale deltas.

Step accumulation

See **Accumulation**.

Straggler

A participant that completes local work later than others. DumbDiLoCo does not wait for stragglers or enforce fixed rounds.

SwiGLU

A gated MLP using `SiLU(gate(x)) * up(x)` followed by a down projection.

Target batch

The nominal number of examples represented by one optimizer update, calculated as microbatch size times accumulation steps.

Token rate

Tokens per second computed by the trainer from input tensor shape or attention-mask sum during the current invocation.

Trainer

The architecture-neutral loop in `trainer.py` that moves batches, executes forward/backward, updates optimizer/scheduler, emits metrics, checkpoints, and calls the coordinator.

Trusted writers

The DumbDiLoCo security assumption: every account with Hub write permission is honest and authorized to influence global weights.

Worker

A DumbDiLoCo participant that trains locally, uploads deltas, and installs newer global weights but does not aggregate outer updates.

Warmup

Initial scheduler steps where the LR rises linearly to its configured starting value.

Related: Core Concepts, Architecture, and DumbDiLoCo.