nanochat-turk:
Efficient Turkish Small LLMs via Verified Synthetic Data
A nanochat Adaptation and Evaluation with Tokenizer Optimizations

BEDAI TEKNOLOJİ A.Ş.
Boğaziçi Teknopark – Kandilli Kampüsü
Kandilli Mah. Rasathane Cad. No:104/13-2, Üsküdar
İstanbul, Türkiye
info@bedai.com.tr

17 January 2026

Abstract

We build a Turkish adaptation of nanochat and study whether verified synthetic data improves the performance of compute-efficient small language models built from a minimal, end-to-end training stack [9]. Our project integrates **Synthetic Data Generation**, **Fine-Tuning / Adaptation**, and **Evaluation & Benchmarking**. We train two Turkish nanochat models (targeting ∼133M parameters each): (i) a baseline trained on Turkish corpora and Turkish instruction data, and (ii) a synthetic-augmented variant trained with additional verified synthetic training samples that we produced. Framed by the recent shift toward efficient SLMs for edge and constrained hardware, we follow a data-centric training recipe and seed SFT with high-quality Turkish instructions before expanding via verified synthetic generation. [7, 13] We evaluate both models on a CETVEL subset of Turkish benchmarks (XNLI-TR, Belebele-TR, PLU-NE, TurkishMMLU) and on held-out synthetic benchmarks designed to stress generalization and robustness. To motivate and de-risk design choices for compute-efficient Turkish micro-models, we also include a controlled tokenizer ablation study at smaller scales (depth $d \in \{2, 4\}$; vocab sizes 32k/64k), evaluating on the same TR suite. This work is part of an ongoing public repository, `nanochat-turkish-synthetic` [17] (https://github.com/nurcunal/nanochat-turkish-synthetic). In the current snapshot, the main model size is 133.0M parameters (d10), with a base pretraining budget of approximately 2.66B tokens (5,075 iterations at batch size 524,288). We generate 800 synthetic items (3 TRAIN + 5 EVAL), and 216 TRAIN items survive strict judging and are used in mid-training/SFT mixes.

1 Introduction

1.1 Small language models (SLMs) and edge efficiency

Small language models (SLMs) are language models in the hundreds-of-millions to few-billion parameter range that trade raw scale for practical deployment: lower latency, smaller memory footprint, and lower energy cost, enabling inference on single GPUs, CPUs, and edge devices. Recent research has shifted from "bigger is always better" toward data-centric recipes and efficient architectures that close capability gaps at small scales, documented in community playbooks and in open releases such as SmolLM3, the Qwen family (including Qwen3), and Meta's Llama 3.x small variants. [7, 8, 15, 12] Meta's MobileLLM further emphasizes sub-billion, on-device language models with efficiency-focused architectural choices, reinforcing the SLM shift toward edge deployment.

[10] This project operates squarely in the SLM regime (∼133M parameters) to keep training and evaluation feasible under limited compute while testing whether verified synthetic data can materially improve Turkish performance. At small scales, gains are driven less by raw parameter count and more by data quality, curriculum, and tokenization. SmolLM2 highlights the impact of large, curated token budgets for SLMs, while SmolLM3 extends this line with long-context and multilingual capability; in parallel, Qwen2.5 and Llama 3.x emphasize improved data mixtures and tokenization for efficient models. [3, 8, 16, 11] Our 133M models are smaller than these 1–3B-class SLMs, but they enable controlled experiments on the role of verified synthetic data and offer a realistic compute footprint for single-node training and evaluation.

Project overview. We build an end-to-end Turkish LLM system by adapting nanochat to support pretraining, mid-training, and supervised fine-tuning (SFT) on Turkish data. The pipeline integrates verified synthetic training data and verified synthetic benchmarks (teacher generation + LLM-judge filtering), and we assess the resulting models on Turkish benchmark suites while running controlled ablations over tokenization and synthetic mix ratios.

1.2 Problem statement and motivation

- **Motivation:** Turkish has fewer high-quality open instruction and benchmark resources than English, and compute budgets are often constrained.
- **Goal:** demonstrate that carefully verified synthetic data can measurably improve small Turkish LLMs under limited compute.
- **Language challenge:** Turkish morphology and agglutination create long, compositional word forms, making tokenizer choice and data coverage critical for small models.
- **Practical constraint:** we target single-node, limited-GPU training, motivating efficiency-first engineering (attention backends, batch sizing, and evaluation throughput).
- **Data scarcity:** a curated seed set and verified synthetic expansion are used to increase coverage while controlling noise and leakage.
- **Context:** the SLM trend emphasizes efficient, deployable models (e.g., SmolLM3, Qwen3, and Llama 3.x 3B-class variants), motivating a Turkish small-model focus rather than scaling parameters indefinitely. [8, 15, 12]

1.3 Contributions

- **Engineering adaptation:** a Turkish fork of nanochat with new dataset/task wrappers, TR training/eval suites, and hardware-friendly attention backend selection for non-Hopper GPUs (specfically a single A100 for Colab compatibility).
- **Synthetic data pipeline:** a verified generation workflow (seed curation, teacher sampling, schema checks, deduplication, and LLM-judge filtering), plus held-out synthetic benchmarks for generalization testing.
- **Evaluation:** benchmarking baseline vs synthetic-augmented model on public TR benchmarks and synthetic held-out sets, with efficiency reporting.

- **Tokenizer ablation (supporting study):** a controlled morphology-aware tokenizer experiment (raw BPE vs segmented variants), evaluating both bpb and downstream accuracy under matched compute.

These contributions aim to deliver a reproducible, compute-feasible Turkish SLM stack and a careful assessment of whether verified synthetic data can substitute for missing high-quality supervised resources.

Ongoing public project. This work is part of an ongoing public repository, `nanochat-turkish-synthetic` [17] (https://github.com/nurcunal/nanochat-turkish-synthetic).

1.4 Success criteria

- **Quality:** improve Turkish benchmark scores vs baseline at comparable compute.
- **Rigor:** verify synthetic data quality (automatic metrics + LLM-judge rubric) and avoid benchmark leakage.
- **Reproducibility:** provide configs, commands, and logs (e.g., W&B runs).
- **Efficiency:** report throughput and memory metrics to reflect edge/limited-hardware deployment constraints. [7]
- **Generalization:** demonstrate improvements on held-out synthetic benchmarks without overfitting to prompt templates.
- **Transparency:** document acceptance rates, failure modes, and data mix ratios for synthetic data.

We treat these criteria as the minimum bar for claiming a meaningful synthetic-data gain in a low-resource Turkish setting.

2 System / Codebase Adaptation (nanochat → Turkish nanochat)

2.1 Overview of the Turkish fork

The Turkish fork extends nanochat with new dataset wrappers, a Turkish training suite, and a Turkish evaluation suite while keeping the core training stack and configuration interface close to upstream [9]. Key additions include Turkish instruction and MCQ dataset adapters in `tasks/`, a TR evaluation harness for a CETVEL subset (XNLI-TR, Belebele-TR, PLU-NE, TurkishMMLU), and entry-point scripts (`turkish.sh`, `tr_base_eval.py`) that mirror the upstream run flow but with TR-specific defaults. We prioritize minimal, auditable changes so that future upstream improvements can be merged without large diffs. Where possible, we rely on the existing nanochat abstractions (dataset streaming, checkpointing, and evaluation hooks) and only extend behavior for Turkish-specific data sources and benchmarking.

2.2 Hardware and runtime constraints

The upstream nanochat scripts are designed for an 8×H100 node. Our project adapts the pipeline to run on constrained hardware (e.g., 1×A100) while retaining the end-to-end workflow. This constraint shapes micro-batch size, sequence length, and evaluation cadence; we rely on gradient accumulation and careful checkpoint scheduling to fit within memory limits while maintaining stable throughput.

We also prioritize attention backends that are robust on non-Hopper GPUs and track runtime metrics (tokens/sec, MFU, wallclock) as first-class outputs for comparing efficiency across runs.

2.3 Attention backend adaptation (FlashAttention vs SDPA)

The Turkish fork implements attention backend selection in `nanochat/gpt.py`:

- Hopper (sm90+): attempt FlashAttention-style kernels [5, 4].
- Otherwise: fall back to PyTorch scaled dot-product attention (SDPA) [14], including GQA-style KV handling [1] and sliding-window masking.

The backend is selected automatically based on GPU capability, with an override via `NANOCHAT_ATTENTION`. On non-Hopper devices, SDPA is used with explicit KV expansion to emulate GQA and with custom attention masks for sliding-window patterns. This keeps the model runnable on commodity GPUs while preserving correctness and providing a clear performance baseline against Hopper-optimized FlashAttention.

Table 1: Observed efficiency on 1×A100 (Colab) from notebook logs (`DSAI585_nanochat_turk.ipynb`).

Run	Depth	Params (M)	Peak VRAM (MiB)	Tok/s (approx.)	Min val BPB ↓
Base pretrain	2	17.17	9,180.83	∼0.90M	1.3182
Base pretrain	4	36.70	11,416.28	∼0.62M	1.0443
Base pretrain	10	133.04	26,394.42	∼0.17M	0.8396
Mid-train (snapshot)	10	133.04	26,390.98	176k	0.5787

2.4 Turkish data pipeline changes

- Pretraining data: FineWeb2-HQ Turkish shards (+ optional train-only extra parquets).
- Dataset tooling: train-only augmentation support while keeping the "last shard = validation" convention.
- Tokenizer: retrain for Turkish runs to avoid subtle mismatches.

We keep Turkish data processing deterministic and auditable: language-specific filtering and normalization are applied consistently across shards, the validation shard is isolated to reduce leakage, and optional extra parquets are merged only into the training split. Tokenizer retraining is treated as first-class configuration to avoid silently mixing mismatched vocabularies across stages. For adaptation, we target Turkish instruction-following and multiple-choice reasoning; this motivates a pipeline that mixes Turkish instruction datasets, CETVEL-subset tasks (XNLI-TR, Belebele-TR, PLU-NE, TurkishMMLU), and a verified synthetic expansion seeded from Turkish prompts. This design aligns with the project's goal of improving low-resource Turkish behavior without relying on large-scale English data.

2.5 Turkish training and evaluation suites

The fork adds a TR evaluation suite (`NANOCHAT_EVAL_SUITE=tr`) and a TR training suite (`NANOCHAT_TRAIN_SUITE=tr`). Our primary evaluation suite uses the task wrappers implemented in `nanochat-master-turkish/tasks/` and loads datasets directly from HuggingFace with explicit identifiers:

- **XNLI-TR**: 3-way NLI; `xnli` (config `tr`); implemented in `tasks/xnli_tr.py`.
- **XCOPA-TR**: 2-way causal reasoning; `cambridgeltl/xcopa` (config `tr`); implemented in `tasks/xcopa_tr.py`.
- **Belebele-TR**: 4-way reading comprehension; `facebook/belebele` (config `tur_Latn`); implemented in `tasks/belebele_tr.py`.
- **PLU-NE**: 4-way next-event prediction; `mcemilg/turkish-plu-next-event-prediction`; implemented in `tasks/plu_tr.py`.
- **MMLU-TR**: multi-domain Turkish MCQ; `malhajar/mmlu_tr-v0.2`; implemented in `tasks/mmlu_tr.py`.

The evaluation suite standardizes prompt templates, multiple-choice formatting, and answer extraction to make scores comparable across models. For training, the TR suite consolidates instruction and MCQ datasets into a unified JSONL conversation format so that the same prompt structure is used throughout pretraining, mid-training, SFT, and evaluation.

3 Data

3.1 Pretraining data

- **Dataset:** FineWeb2-HQ Turkish shards [2].
- **Optional augmentation:** FineWeb2-HQ `tur_Latn` parquets (train-only) [6].
- **Validation:** held-out last shard (per nanochat convention).

We stream and shuffle shards deterministically, track corpus statistics (tokens, characters, document counts), and avoid cross-split contamination by reserving the final shard as validation. Optional `tur_Latn` augmentation is introduced only into the training split and recorded separately so that we can report the effective pretraining mix and its impact on downstream metrics. Tokenizer training downloads the first 8 shards (approximately 2B characters, roughly 800MB compressed) and begins background download of the full pretraining corpus. The training script queries the dataset API and caps the download at 256 shards by default (or fewer if the dataset has fewer shards; the dataset currently exposes 1822 shards per the script comment). Each shard is approximately 250M characters ($\sim$100MB compressed). The optional train-only augmentation pulls the first 12 `tur_Latn` parquets by default.

3.2 Mid-training and SFT data (baseline)

- **Turkish instruction datasets:** `tasks/turkish_instruct.py` wrappers (schema-robust).
- **SFT seed set:** $\sim$100 high-quality Turkish instruction examples to anchor SFT and prompt templates, aligned with OpenAI guidance that 50–100 examples often yield measurable gains. [13]
- **Turkish MCQ mix-in (CETVEL subset):** TurkishMMLU (`tasks/mmlu_tr.py`), PLU-NE, and Belebele-TR; plus NLI from XNLI-TR.
- **Persona data:** optional Turkish identity conversations JSONL (oversampled).

We balance instruction and MCQ data to avoid overfitting to multiple-choice styles, cap oversampling of small persona datasets, and deduplicate across sources to reduce prompt leakage. All mid/SFT examples are normalized to the same conversation schema (roles + content) so that the model sees a stable format across stages. **Fine-tuning / adaptation.** We fine-tune a Turkish generative model using supervised fine-tuning (SFT) as the primary alignment step; this is appropriate for low-resource Turkish instruction-following and allows direct control over the training distribution. We emphasize domain adaptation to Turkish instructional dialogue and short-form reasoning rather than domain-specific professional verticals, keeping the model broadly useful for Turkish general-purpose assistants. This is consistent with training a model from scratch under limited compute and evaluating on a CETVEL subset that includes XNLI-TR, Belebele-TR, PLU-NE, and TurkishMMLU. We note that parameter-efficient methods (LoRA/QLoRA) remain viable for later extensions but are not required for the current pipeline because the target model size is already small enough to fine-tune end-to-end; domain-specific adaptation (e.g., law) is deferred to future work pending dedicated datasets and evaluation.

Dataset identifiers used (baseline mid-train/SFT). From the implemented task wrappers in `nanochat-master-turkish/tasks/`, our baseline mid-train and SFT stages use:

- **Instruction**: `suayptalha/Sungur-Dataset` and `AlicanKiraz0/Turkish-SFT-Dataset-v1.0` (best-effort schema conversion in `tasks/turkish_instruct.py`).
- **MCQ mix-in**: `xnli` (config `tr`), `cambridgeltl/xcopa` (config `tr`), `facebook/belebele` (config `tur_Latn`), `mcemilg/turkish-plu-next-event-prediction`, and `malhajar/mmlu_tr-v0.2`.

We report row counts and mix ratios for the current SynAug snapshot; larger-scale generation remains future work.

3.3 Synthetic data generation (TRAIN)

3.3.1 Design goals

- high-signal, learnable Turkish instruction and MCQ samples
- diversity (topics, styles), controlled length, safe content
- strict separation from benchmark test items (no leakage)

We target a balanced mix of task types (instruction following, short-form QA, and MCQ reasoning), with both single-turn and short multi-turn variants. Topic coverage is intentionally broad (news, everyday tasks, STEM, civic knowledge) to reduce overfitting to narrow domains while remaining feasible for a small model to learn. We begin from a curated seed pool of $\sim$100 Turkish instructions/questions and expand with teacher-model generations; the seed size follows OpenAI's SFT guidance (50–100 high-quality examples) and serves as an anchor for verification and prompt templating. [13]

3.3.2 Generation process (teacher model)

We define prompt templates per category (instruction, MCQ, and short multi-turn) and sample multiple candidates per seed at varied temperatures (e.g., 0.3 and 0.8) to balance correctness and diversity. Candidates must parse into the nanochat conversation schema, pass Turkish language checks, and include explicit answer keys for MCQ items. We log per-sample metadata (seed ID,

template type, temperature, judge scores) to support later audits and ablation studies. We follow an instruction-generation paradigm inspired by self-instruct style bootstrapping [19], and we validate outputs with a separate LLM-judge rubric [20] to reduce noise and bias. **Synthetic data rationale.** We use a teacher model to generate Turkish instructions, MCQ items, and short dialogues because high-quality labeled Turkish data is scarce. Teacher prompting and self-instruct style expansions give controllable coverage, while verification (automatic + LLM-judge) keeps noise bounded. We treat synthetic data as a targeted augmentation rather than a full replacement for real Turkish corpora. Alongside teacher-generated synthetic data, we also treat morphology-aware segmentation as a deterministic *data augmentation* view of the same corpus: segmenting text into morpheme-respecting units creates an alternate training signal that can improve sample efficiency for Turkish. This is not synthetic data generation in the teacher-model sense, but it strengthens the data-augmentation story and complements the synthetic pipeline.

3.3.3 Implementation (replicable scripts)

To make synthetic generation and verification reproducible, we implement two standalone scripts in the Turkish nanochat fork:

- `nanochat-master-turkish/scripts/synthetic_generate_openai.py`: uses the OpenAI Chat Completions API as a *teacher model* to generate Turkish synthetic data via prompting (self-instruct-style templating and controlled sampling). It emits strict nanochat CustomJSON JSONL (user/assistant alternating messages; no `system` role) and supports both synthetic TRAIN styles (SungurTR-like, TurkishSFTV1-like, and Turkish MCQ training in an MMLU-TR-like style) and held-out synthetic EVAL benchmark-like sets (XNLI-style NLI, Belebele-style RC, Turkish-PLU-style next-event prediction, and TurkishMMLU-style MCQ). For reproducibility, it writes a `.meta.json` sidecar logging model identifier, temperature, and dataset/benchmark links used as format references.

- `nanochat-master-turkish/scripts/synthetic_judge_gemini.py`: uses Gemini as an *LLM-as-judge* to validate synthetic item quality against a strict rubric aligned with the course requirement to "validate synthetic data quality" and to build a rigorous evaluation suite. The judge scores each item on fluency (TR), usefulness for small models, schema correctness, safety/no-PII, novelty/leakage, and (for MCQ) answer correctness. It outputs `accepted.jsonl`, `rejected.jsonl`, and `judge_report.json` (per-item scores + acceptance rate), enabling threshold ablations (loose vs strict) and transparent reporting of failure modes.

The exact teacher prompt templates and the judge rubric are embedded directly in these scripts (as strings) for reproducibility; we also reproduce the rubric explicitly in this report (Sec. 4.3.2). API keys are provided via environment variables (`OPENAI_API_KEY` and `GEMINI_API_KEY`); we do not commit keys to the repository.

3.3.4 Data format

All synthetic data is stored in nanochat conversation JSONL format: `["role":"user","content":"..."],"role":"assistant","content":"..."]`. MCQ synthetic samples use the assistant content as the gold letter (`A/B/C/D`). We optionally attach a lightweight metadata sidecar (seed ID, template, temperature, judge score) to support filtering and auditing without changing the core training format.

3.3.5 Generation configuration (from `*.meta.json` sidecars)

Each generated file under `/DSAI585-NureddinCuneyd-Unal/synthetic` has a `.meta.json` sidecar that records the teacher-model settings and constraints. From the current snapshot (2026-01-17), the synthetic artifacts were generated with:

- **Teacher model label:** `GPT-5.2 Thinking` across all synthetic datasets/benchmarks in this repo snapshot.
- **Sampling temperature:** 0.7 (fixed across files in this snapshot).
- **Seeds:** recorded per-file for traceability (e.g., XNLI-like uses seed 123; Belebele-like 777; PLU-like 20260117; TurkishMMLU 909; MMLU-TR train 1313; TurkishSFTV1 42). Note that API generation is not fully deterministic; seeds are treated as metadata for experiment bookkeeping.
- **Global constraints (shared):** Turkish-only; strict nanochat CustomJSON schema; no translation/paraphrase/copy from public benchmarks; no PII; no explicit sexual content; no hate/harassment; no self-harm; no illegal instructions; no extremist content; concise and unambiguous prompts suitable for small-model learnability.

Difficulty settings. For MCQ-style synthetic files that include explicit difficulty planning, the sidecars specify a 30/50/20 split: 30% easy, 50% medium, 20% hard (e.g., `synthetic_train_mmlu_tr_100.jsonl.meta.json` and `synthetic_eval_turkishmmlu_100.jsonl.meta.json`). For benchmark-like synthetic EVAL sets, sidecars also include **stressors** (robustness variants): XNLI-like reports 12 stressor items (∼12%), Belebele-like 13 (∼13%), and PLU-like 13 (∼13%).

Task-specific generation details.

- **XNLI-like (3-way NLI):** label map {A=entailment, B=neutral, C=contradiction}; balanced label counts A/B/C = 34/33/33; stressors = 12; fixed prompt template with `Öncül`/`Hipotez` and letter-only answers (`synthetic_eval_xnli_100.jsonl.meta.json`).
- **Belebele-like (4-way RC):** passages constrained to 4–10 sentences; average passage sentence estimate 8.35; stressors = 13; fixed `Parça`/`Soru` template and letter-only answers (`synthetic_eval_belebele_like_100.jsonl.meta.json`).
- **PLU-like (next-event prediction, 4-way):** strict `Hedef`/`Adım`/`Bir sonraki adım` structure; stressors = 13; `procedure_families`=10 indicating multiple procedure domains (e.g., cleaning, cooking, scheduling) (`synthetic_eval_plu_100.jsonl.meta.json`).
- **TurkishMMLU-like (4-way MCQ):** subject coverage spans 9 school-style categories (Math/Physics/Chemistry/Biology/History/Geography/Literature/Philosophy/Grammar) with 30/50/20 difficulty plan; this file explicitly replaces an earlier "MCQ sprinkle" plan while enforcing the project-wide 4-option constraint (`synthetic_eval_turkishmmlu_100.jsonl.meta.json`).
- **MMLU-TR-like TRAIN (4-way MCQ):** includes both subject counts and answer-letter counts (A/B/C/D = 19/31/26/24) to detect positional bias; 30/50/20 difficulty plan (`synthetic_train_mmlu_tr_100.jsonl.meta.json`).
- **TurkishSFTV1-like TRAIN (instruction):** 100 rows with 20 multi-turn rows, 80 two-turn rows, average 2.4 messages/row; topic notes include writing/rewrites, planning/productivity, light reasoning, and minimal code help (`synthetic_train_turkishsftv1_100.jsonl.meta.json`).

- **SungurTR-like TRAIN (instruction+reasoning):** think-block fraction estimate 0.6 and a coarse topic distribution (math-heavy in this snapshot: 40/100 tagged as math by heuristic; 10 summarization; 9 writing; 10 logic) with explicit schema validation checks (`synthetic_train_sungurtr_100.jsonl.meta.json`).

3.3.6 Synthetic artifacts in this repository (current snapshot)

We store generated artifacts under `synthetic/`: `synthetic_datasets/` (TRAIN JSONL + `.meta.json`), `synthetic_benchmarks/` (EVAL JSONL + `.meta.json`), and `llm-judge/` (LLM-judge reports with accepted/rejected partitions). The roadmap is: (i) generate candidates with the teacher script; (ii) judge them with the LLM rubric; (iii) reconstruct accepted sets via `build_accepted_from_judged.py`; and (iv) materialize train mixes and validation files (including `val_synth_only.jsonl`) with `build_mix_files.py`. In the current snapshot, we generated 800 synthetic items across eight JSONL files (3 TRAIN + 5 EVAL, each 100). After strict judging, 216 training items survived (72 + 51 + 93) and were consolidated into `val_synth_only.jsonl` (216 lines, see `/Users/nurcunal/Downloads/val\_synth\_only.jsonl`). These 216 accepted TRAIN samples are used in both mid-training and SFT mixes; accepted EVAL sets are held out. Table 2 summarizes the per-source survival counts.

Why these are *datasets* and *benchmarks* (not "add-ons"). We treat each synthetic artifact as a legitimate new dataset/benchmark because it is (i) generated as a self-contained labeled corpus in a strict, trainable schema (nanochat CustomJSON), (ii) accompanied by auditable `.meta.json` sidecars documenting generation constraints and difficulty/stressor settings (Sec. 3.3.5), and (iii) subjected to independent quality review by a separate model using an explicit rubric with accept/reject policy (Sec. 4.3.2; Tables 2–3). For synthetic **TRAIN**, only items that pass validation are admitted into training mixes; for synthetic **EVAL**, the sets are frozen and held out (never used for training), making them bona fide benchmarks for cross-distribution generalization. This directly satisfies the course requirement for **Synthetic Data Generation** (generate with a teacher model and validate quality) and strengthens **Evaluation & Benchmarking** (rubric-based judging + robustness stressors + ablations) [19, 20].

Table 2: Synthetic artifacts and survival counts (2026-01-17 snapshot).

Artifact	Kind	Total	Accepted	Rate	Notes
`synthetic_train_sungurtr_100.jsonl`	TRAIN	100	72	0.72	Instruction + reasoning (SungurTR-like)
`synthetic_train_turkishsftv1_100.jsonl`	TRAIN	100	51	0.51	Instruction (TurkishSFTV1-like)
`synthetic_train_mmlu_tr_100.jsonl`	TRAIN	100	93	0.93	MCQ (MMLU-TR-like)
`synthetic_eval_xnli_100.jsonl`	EVAL	100	74	0.74	NLI (XNLI-like)
`synthetic_eval_belebele_like_100.jsonl`	EVAL	100	48	0.48	RC (Belebele-like)
`synthetic_eval_plu_100.jsonl`	EVAL	100	70	0.70	Next-event (PLU-like)
`synthetic_eval_turkishmmlu_100.jsonl`	EVAL	100	5	0.05	MCQ (TurkishMMLU-like)
`synthetic_eval_xcopa_like_100.jsonl`	EVAL	100	45	0.45	Causal (XCOPA-like)
Total (TRAIN)	TRAIN	300	**216**	**0.72**	Used in mid-training and SFT mixes
Total (EVAL)	EVAL	500	**242**	**0.48**	Held out (never used for training)

All synthetic generations and judge artifacts in this repository are stored under: `/DSAI585-NureddinCuneyd-Unal/synthetic`.

Table 3: Qualitative examples from generated synthetic artifacts and LLM-judge accept/reject decisions (truncated).

Artifact	Example (generated / accepted)	Example (rejected, with reason type)
`synthetic_train_sungurtr_100.jsonl`	"Aşağıdaki parçayı 2–3 maddede özetle: ..." → madde madde özet	—
`synthetic_train_turkishsftv1_100.jsonl`	"Aktif hatırlama yöntemini 4 maddeyle anlat." → 4 maddelik açıklama	—
`synthetic_train_mmlu_tr_100.jsonl`	"Deprem riski yüksek bölgelerde en etkili önlem hangisidir? ..." → `C`	—
`synthetic_eval_xnli_100.jsonl`	"Öncül: ... Hipotez: ... ilişki hangisidir?" → `A/B/C`	—
`synthetic_eval_plu_100.jsonl`	"Hedef: ... / Adım: ... / Bir sonraki adım: ..." → `A/B/C/D`	—
`synthetic_eval_belebele_like_100.jsonl`	"Parça: ... Soru: ..." → `A/B/C/D`	—
`synthetic_eval_turkishmmlu_100.jsonl`	"Isı sığası ... Sıcaklık artışı kaç birim olur? ..." → `A`	—
`synthetic_train_sungurtr_100_judged.jsonl`	Accepted (example): "Aşağıdaki parçayı 2–3 maddede özetle: ... Bursa'de ..." → kısa madde madde özet	Rejected (example): "300 km'lik bir yolu önce 98 km/s, sonra 82 km/s..." ⇒ PHYSICAL_REALISM (gerçek-dışı birim/hız)
`synthetic_train_mmlu_tr_100_judged.jsonl`	Accepted (example): "Aşağıdaki kelimelerden hangisi doğru yazılmıştır? A) her hangı B) herhangi C) herhangı D) her-hangi" → `B`	Rejected (example): seçeneklerde `0.6666666666666666` gibi aşırı ondalık ⇒ FLOAT_PRECISION_ARTIFACT
`synthetic_eval_xnli_100_judged.jsonl`	Accepted (example): "Öncül: Emre, kafedeki sergiyi gezdi... / Hipotez: Emre sergiyi gezdi." → `A`	Rejected (example): "Aslı bugün metro istasyonue gitmedi..." ⇒ MORPHOLOGY_BROKEN_NOUNS (`istasyonue`)
`synthetic_eval_plu_like_100_judged.jsonl`	Accepted (example): "Hedef: çalışma masası temizliğini hızlıca bitirmek / Adım: ..." → `C`	Rejected (example): "Hedef: Hatay'de ..." ⇒ MORPHOLOGY_PROPER_NOUNS (`Hatay'de`→`Hatay'da`)
`synthetic_eval_belebele_like_100_judged.jsonl`	Accepted (example): RC parçası + "Soru: Sorun neden ortaya çıktı?" → `A`	Rejected (example): "Kocaeli'deki otogarde ... atölyee ..." ⇒ MORPHOLOGY_LOCATIVE_HARD_ERROR (`otogarde`)
`synthetic_eval_turkishmmlu_100_judged.jsonl`	Accepted (example): "DNA'nın temel görevi hangisidir? ..." → `A`	Rejected (example): seçeneklerde `33.333333333333336` ⇒ FLOAT_FORMATTING / FLOAT_PRECISION_ARTIFACT

3.4 Synthetic benchmark(s) (EVAL)

We construct held-out synthetic benchmark-like evaluation sets in the same nanochat CustomJSON JSONL format as training, but with disjoint seeds and stricter filtering. In the current snapshot, the synthetic EVAL suite is MCQ-focused and consists of 5×100 items: `synthetic_eval_xnli_100.jsonl`, `synthetic_eval_belebele_like_100.jsonl`, `synthetic_eval_plu_100.jsonl`, `synthetic_eval_turkishmmlu_100.jsonl`, and `synthetic_eval_xcopa_like_100.jsonl`. These sets are never used for training and are frozen early to minimize leakage risk. We construct held-out synthetic benchmarks from disjoint seeds and templates, with stricter filtering than the training set, and we freeze them early to prevent leakage. These benchmarks are used to assess generalization beyond the prompt patterns seen in training and to check whether synthetic gains transfer to unseen topics and styles. In addition to the public TR benchmarks (XNLI-TR, XCOPA-TR, Belebele-TR, and PLU-NE), we evaluate on synthetic benchmark-like MCQ sets using standard **automatic metrics** (categorical letter accuracy) and report **LLM-judge rubric** summaries as complementary evidence (e.g., for instruction-style held-out sets where exact-match is not meaningful). This directly addresses the course requirement for **Evaluation & Benchmarking** via automatic metrics, LLM-judge with rubric, and ablations/robustness checks (e.g., strict vs loose judge thresholds; paraphrase and distractor stressors in synthetic EVAL).

4 Methods

4.1 Model configuration

- **Target size:** ∼133M parameters (two models), intentionally in the SLM regime for efficient training/inference and edge-adjacent deployment. [7, 8, 15, 12]
- **Tokenizer:** retrained Turkish tokenizer (vocab size 65,536 / 64k); additionally, we run a controlled tokenizer ablation (Sec. 5.2) to justify the tokenizer choice used in the main baseline vs synthetic-aug comparison.
- **Training stages:** pretrain → mid-train → SFT.

The model is a decoder-only Transformer [18] with rotary embeddings, QK normalization, RMSNorm without learned parameters, untied token embedding and output projection weights, ReLU^2 MLPs, and no bias terms in linear layers. Grouped-query attention and optional sliding-window patterns are used to improve efficiency at small scale and to keep inference feasible on limited hardware. For base pretraining, the configuration is derived from `scripts/base_train`: model dimension scales as $d_{\text{model}} = \text{depth} \times 64$, head dimension defaults to 128, max sequence length is 2048, and the default attention window pattern is `SSSL`. The notebook logs confirm a 133,038,100-parameter configuration with 5,075 iterations computed from the target data:param ratio, and smaller ablation configs at 17,170,436 parameters (depth 2) and 36,700,168 parameters (depth 4).

Table 4: Tokenizer ablation micro-model configs (from notebook logs).

Depth	d_{model}	Heads	Seq len	Vocab	Params (M)
2	128	1	2048	65,536	17.17
4	256	2	2048	65,536	36.70

4.2 Training pipeline

4.2.1 Baseline model

The current baseline pipeline follows the `turkish.sh` script: base pretraining runs `scripts.base_train` with `-depth=10`, `-target-param-data-ratio=20`, default max sequence length 2048, default total batch size 524,288 tokens, and checkpointing every 1000 steps. CORE evaluation is disabled for Turkish runs (`-core-metric-every=-1`), while val bpb is retained as a language-agnostic signal on the Turkish validation shard. The notebook logs also record the computed iteration counts used for target data:param ratio schedules (e.g., 5,075 iterations for the 133M configuration). We follow a data-centric staging recipe common in small-model playbooks: broad pretraining, domain/skill-focused mid-training, then SFT for instruction following. [7] Optimization follows the nanochat defaults: a dual-optimizer setup with AdamW for embeddings, output head, and scalar parameters, and Muon for matrix weights, with model-dimension-scaled learning rates and separate parameter groups. Learning rates, betas, and weight decay are logged in the run configs/notebook, together with checkpoint cadence, evaluation frequency, and throughput metrics.

4.2.2 Synthetic-augmented model

Same architecture and compute budget as baseline, but with verified synthetic training data mixed in. Mixing ratios are 0/25/50/75/100% and are injected at both mid-training and SFT by

materializing mix files with `synthetic/build_mix_files.py`. Mid-stage mixes use $n_{\text{train}} = 50{,}000$ and $n_{\text{val}} = 2{,}000$; SFT mixes use $n_{\text{train}} = 20{,}000$ and $n_{\text{val}} = 2{,}000$ (with `sft_val_size`=2,000). Synthetic-only validation (`val_synth_only.jsonl`) contains 216 accepted items. Mix construction is deterministic and fully specified in code: `build_mix_files.py` loads the judge-accepted synthetic TRAIN pool and original Turkish tasks (SungurTR, TurkishSFTV1, XNLI-TR, MMLU-TR for mid; TurkishSFTV1+MMLU-TR for SFT), then samples with weights proportional to dataset sizes and normalizes all conversations into strict CustomJSON (system messages folded into the first user, consecutive same-role messages merged, and assistant-first samples rejected). For each ratio r, it draws n_{orig} and n_{syn} using a seed tied to r, shuffles the mix, writes `mix_`$\{r\}$`syn_train.jsonl`, and emits both `val_original.jsonl` and `val_synth_only.jsonl` plus a manifest of counts and sources. This makes the mix ratios reproducible and auditable without external API calls. We keep the total number of training tokens constant between baseline and SynAug runs to isolate the impact of synthetic data quality rather than total compute. Mixing is evaluated at multiple ratios and, where feasible, injected both in mid-training (to shape general capability) and in SFT (to sharpen instruction following). Leakage checks are applied against public TR benchmarks before any synthetic sample is admitted.

4.3 Synthetic data verification (quality control)

4.3.1 Automatic metrics and filters

- schema validity (JSON, roles, non-empty fields)
- Turkish language compliance (lang-id / heuristics)
- length bounds and trivial-response filtering
- dedup + near-dup against seeds and public benchmark items (leakage prevention)
- diversity summaries (distinct-n / minhash buckets)
- safety heuristics (profanity/toxicity/PII screening)

These filters are applied in a staged pipeline: fast schema and language checks first, then deduplication and length constraints, followed by diversity and safety screens. We record filter pass/fail counts to analyze where synthetic candidates are lost and to tune prompt templates accordingly.

4.3.2 LLM-judge rubric (second model)

- fluency/naturalness (TR)
- instruction clarity
- MCQ correctness and distractor quality
- usefulness for training small models (clarity, not overly long)
- safety/no PII

In our code, the judge model defaults to `gemini-1.5-pro` (configurable via `GEMINI_MODEL`) and uses a strict acceptance threshold: accept only if all non-MCQ rubric scores are ≥ 4 and, for MCQ items, the MCQ correctness score is ≥ 4 (Sec. 4.3.2). We calibrate judge thresholds on a small

manually reviewed set and keep both strict and loose variants to study the accuracy–diversity trade-off. Judge prompts and scoring rubrics are included in the appendix for reproducibility, along with accepted/rejected examples (Table 3).

These verification steps implement the project's quality guarantee for synthetic data and directly support the evaluation rubric (fluency, correctness, usefulness, and safety) used later in benchmarking.

Rubric used in code (Gemini-as-judge). Our judge implementation (`nanochat-master-turkish/scripts/synthetic_judge_gemini.py`) scores each synthetic item on a 0–5 scale with the following fields:

- `turkish_fluency`: grammaticality and naturalness in Turkish
- `clarity_usefulness_small_llm`: clarity + learnability for a small model (concise, unambiguous)
- `format_schema_correct`: strict nanochat CustomJSON format (alternating `user`/`assistant`; no `system`)
- `safety_no_pii`: no unsafe content, no personal data
- `novelty_no_memorization`: novelty and low risk of benchmark leakage
- `mcq_correctness_if_applicable`: for MCQ items, verify the gold letter is actually correct (otherwise set to 5)

We accept an item only if all non-MCQ scores are $\geq$ 4 and, for MCQ items, `mcq_correctness_if_applicable` is ≥ 4; otherwise it is rejected. The script emits `accepted.jsonl`, `rejected.jsonl`, and `judge_report.json` (per-item scores + acceptance rate), enabling strictness ablations (loose vs strict) and transparent reporting of failure modes.

Table 5: LLM-judge rubric definition and acceptance policy (used for all synthetic TRAIN/EVAL validation).

Rubric dimension (0–5)	Score guidance (what we require for acceptance)
`turkish_fluency`	5: fully natural Turkish; 4: minor typos/suffix issues but readable and not misleading; $\leq$3: broken morphology or non-Turkish fragments that harm meaning.
`clarity_usefulness_small_llm`	5: clear prompt, unambiguous, high-signal answer; 4: mostly clear with minor verbosity; $\leq$3: ambiguous, underspecified, or low training value (e.g., trivial/rambling).
`format_schema_correct`	5: strict CustomJSON (alternating user/assistant; no system; non-empty strings); $\leq$3: schema violations or formatting that breaks parsers.
`safety_no_pii`	5: no PII, no unsafe content; $\leq$3: any PII or disallowed safety category triggers rejection.
`novelty_no_memorization`	5: clearly novel; 4: plausible novelty with low leakage risk; $\leq$3: suspected paraphrase/translation/copy of public benchmark or seed-like memorization.
`mcq_correctness_if_applicable`	For MCQ items: 5: gold letter correct; 4: correct with minor phrasing issues; $\leq$3: incorrect/ambiguous label. For non-MCQ: set to 5.
Acceptance policy	Accept iff all non-MCQ scores are $\geq$ 4 and (if MCQ) `mcq_correctness_if_applicable`≥ 4; otherwise reject.

Course-requirements alignment (explicit). This teacher+judge pipeline is designed to provide visible evidence for the DSAI585 rubric: **Synthetic Data Generation** (teacher prompting/self-instruct-style generation + quality validation) and **Evaluation & Benchmarking** (automatic metrics + LLM-judge rubric + robustness stressors and strictness/mix-ratio ablations) [19, 20].

5 Experiments

5.1 Experiment matrix

Table 6: Synthetic mix ablation grid (mid/SFT).

Run	Model	Training data	Synthetic mix	Notes
E0	Baseline	Turkish (vanilla)	0%	baseline pipeline
E1	SynAug	Turkish + synthetic	25%	mid+SFT mix
E2	SynAug	Turkish + synthetic	50%	mid+SFT mix
E3	SynAug	Turkish + synthetic	75%	mid+SFT mix
E4	SynAug	Turkish + synthetic	100%	mid+SFT mix

Each run shares architecture and total token budget; only the data mixture and verification strictness vary. Where possible we repeat runs with a second seed to estimate variance, and we tag all runs with W&B IDs for traceability.

5.2 Tokenizer ablation study (supporting adaptation/efficiency evidence)

Why the ablations (Smol Training Playbook motivation). Following practical LLM training guidance that emphasizes small, controlled ablations to validate end-to-end pipelines under compute constraints (e.g., the Smol Training Playbook $\text{smol}_t raining_p laybook), we prioritize a factorial design over (i) tokenization condition (baseline vs. segmented), (ii) se$
$base dvs. neural), (iii) vocabulary size (32k vs. 64k), and (iv) shallow model depths (e.g., d} \in \{2, 4\})$ to isolate tokenizer effects before scaling. Turkish is morphologically rich and agglutinative; token boundaries that cut across morphemes can make learning less sample-efficient for small models. This ablation is a supporting adaptation/efficiency study and does *not* replace our synthetic-data component; it serves as a justified design decision for the main experiments.

5.2.1 Permutation ablations

We run (and continue to run) a permutation grid over segmentation backend, vocabulary size, and model depth. For segmented variants, we apply morphology-aware preprocessing before BPE training, then map tokens back to standard text for training and evaluation. This isolates segmentation effects from downstream training configuration and ensures that differences in bpb or accuracy reflect tokenizer choices rather than unrelated hyperparameter changes. Implementation details follow our tokenizer pipeline (as summarized in `tokenizer-report.tex`): we segment the same raw Turkish corpus using TRMorph (rule-based) and TurkishDelightNLP (neural), then train RustBPE on the segmented text so merges cannot cross morpheme boundaries. The segmentation and normalization code is implemented in `MorphTokenizer/tr_morph_segment.py`, with inspection utilities in `parquet_viewer.py` and tokenization diagnostics (fertility, continuation ratio) in `assess_tokenization_metrics.py`. Segmented datasets are written to

`MorphTokenizer/added/output_files/segmentation/<backend>/segmented_datasets/` for reproducible training runs. Step-by-step segmentation instructions are documented in the tokenizer READMEs (`README.txt` and `src/README.txt`) bundled in the Drive code cache under `nanochat-repo-cache/src/` (see Reproducibility for the download link). Because segmentation creates a transformed view of the same corpus, we also describe it as a lightweight, deterministic augmentation that injects morphological structure into training without relying on a teacher model; this complements (but does not replace) our synthetic generation pipeline.

Table 7: Each row is a single training run under identical settings except for segmentation + vocabulary size (and model depth).

Depth d	Params	Train tokens	Vocab	Segmentation	CETVEL subset macro ↑
2	20M	160M	32k	None (raw)	0.2107
2	20M	160M	32k	TRMorph	0.2412
2	20M	160M	32k	TurkishDelightNLP	0.2336
2	37M	294M	64k	None (raw)	0.2153
2	37M	294M	64k	TRMorph	0.2443
2	37M	294M	64k	TurkishDelightNLP	0.2321
4	20M	160M	32k	None (raw)	0.2412
4	20M	160M	32k	TRMorph	0.2458
4	20M	160M	32k	TurkishDelightNLP	0.2427
4	37M	294M	64k	None (raw)	0.2030
4	37M	294M	64k	TRMorph	0.2564
4	37M	294M	64k	TurkishDelightNLP	0.2122

5.2.2 Mini CETVEL sanity-check

Before running the full ablation grid on the entire CETVEL suite, we validate the end-to-end integration using a representative CETVEL task. Table 7 reports results on `CETVEL subset macro`.

Table 8: Tokenizer diagnostics and end-of-training bits-per-byte (BPB). Fertility is the average number of tokens per word; continuation ratio is the fraction of tokens that are not word-start tokens. BPB values are taken from the Colab demo notebook logs for the 32k runs ("val bpb"); lower is better.

Vocab	Condition	Fertility (tokens/word) ↓	Continuation ratio ↓	Val BPB ↓
32k	Baseline BPE (raw)	1.643	-	0.8570
32k	TRMorph-seg	2.357	-	0.8171
32k	TurkishDelightNLP-seg	1.831	-	0.8774
64k	Baseline BPE (raw)	1.643	-	0.4535
64k	TRMorph-seg	2.295	-	0.2975
64k	TurkishDelightNLP-seg	1.711	-	0.8171

To interpret benchmark deltas, we report intrinsic tokenization diagnostics.

Fertility. We define *fertility* as the average number of tokens per word.

Continuation ratio. We define *continuation ratio* as the fraction of tokens that are *not* word-start tokens. For GPT-style BPE tokenizers, a practical proxy is the fraction of tokens that do not begin with the tokenizer's whitespace/word-start convention (implementation-dependent).
How we use these results. We select the tokenizer setting that provides the best accuracy–efficiency trade-off at small scale, then carry that tokenizer choice into the final ∼133M baseline vs synthetic-aug comparison to avoid confounding effects.

5.3 Baselines: other open(-ish) models

We list a small set of publicly available models on overlapping Turkish benchmarks (not run in the current snapshot). Baseline selection prioritizes size-matched SLMs (1–3B) with multilingual coverage, then larger open models as reference points when compute permits. When run, we use consistent prompts and decoding settings across baselines to reduce evaluation artifacts.

- **Size-matched SLMs (1–3B):** SmolLM3-3B, Qwen2.5-3B, Llama 3.2-3B [8, 16, 12]
- **Multilingual reference baselines (7–8B):** Llama 3-8B-Instruct, Mistral-7B-Instruct, Qwen2.5-7B-Instruct [11, 16]
- **Turkish-specialized baselines:** none included in the current snapshot due to compute constraints; will be added in a follow-up evaluation pass.

5.4 Evaluation protocol

5.4.1 Primary Turkish benchmarks (TR suite)

We evaluate with a Turkish multiple-choice benchmark suite implemented in `nanochat-master-turkish/tasks/`:

- **XNLI-TR**: `xnli` (config `tr`), 3-way accuracy.
- **XCOPA-TR**: `cambridgeltl/xcopa` (config `tr`), 2-way accuracy.
- **Belebele-TR**: `facebook/belebele` (config `tur_Latn`), 4-way accuracy.
- **PLU-NE**: `mcemilg/turkish-plu-next-event-prediction`, 4-way accuracy.
- **MMLU-TR**: `malhajar/mmlu_tr-v0.2`, categorical accuracy.

Evaluation settings follow the training scripts: during periodic in-training evaluation we cap per-task evaluation with `-eval-metrics-max-problems=1024` (default in `scripts/chat_sft.py`) for speed. From notebook logs we observe that XCOPA-TR is evaluated on its full test set (500 items), Belebele-TR on 900 items, and XNLI-TR either on a 1024-item cap during training or on the full 5010-item test set for final evaluation (e.g., `1742/5010` correct ⇒ 34.77%). We use consistent multiple-choice templates with letter-only outputs and normalize whitespace and casing when extracting answers. Where datasets provide context paragraphs, we keep the prompt length within the model's context window and report truncation strategies in the appendix.

5.4.2 CETVEL subset (tokenizer ablation)

CETVEL is a unified Turkish benchmark suite spanning 23 tasks. For the tokenizer ablation, we run a representative CETVEL subset to validate the end-to-end pipeline under limited compute and report a macro-average across the subset ("CETVEL subset macro"). We report CETVEL subset results and tokenizer diagnostics as tables to keep the main text compact; learning dynamics are illustrated separately via W&B curves (Sec. 6.5). While CETVEL is used primarily for tokenization ablations (compute-feasible and broad), our main benchmark suite for baseline vs SynAug reporting is the TR suite above; there is partial task overlap (e.g., PLU-NE-style and Turkish MCQ), but we keep results clearly separated by suite.

5.4.3 Synthetic benchmarks (held-out)

- synthetic MCQ accuracy
- synthetic instruction benchmark: held-out loss and/or LLM-judge rubric on completions

Synthetic evaluation uses a fixed held-out set with disjoint seeds and prompts. For instruction-style items, we report both loss-based metrics and judge scores to capture fluency and instruction adherence beyond exact-match evaluation.

5.4.4 Robustness, hallucination, and safety checks

We include targeted probes for hallucination and safety by sampling model outputs on ambiguous or low-evidence prompts and checking for unsupported claims, unsafe content, or leakage from benchmark items. These checks are qualitative for now but use the same LLM-judge rubric to ensure consistent scoring. We also report failure modes from the synthetic-data filters (e.g., non-Turkish outputs, incorrect MCQ labels) as a proxy for robustness of the generation pipeline itself.

5.4.5 Statistical reporting

- report absolute scores and deltas vs baseline
- bootstrap confidence intervals over examples
- record compute/time (tokens/sec, MFU, wallclock)
- report latency/memory where feasible to capture SLM deployment trade-offs

When feasible, we run at least two random seeds for core configurations to estimate variance and avoid over-interpreting small deltas.

5.5 Ablations

Table 9: Ablations planned for synthetic data.

Ablation axis	Values	Metric(s)	Hypothesis
Synthetic mix ratio	0/25/50/75/100%	TR benchmark acc, syn acc	more (verified) synthetic helps
Judge strictness	loose vs strict	TR benchmark acc, syn acc	strict improves signal, may reduce diversity
Teacher temperature	0.3 vs 0.8	syn quality, acceptance rate	higher temp increases diversity but may reduce correc

Each ablation varies one axis at a time while keeping the architecture and total token budget fixed, enabling clearer attribution of gains to data quality or diversity rather than compute scale.

Table 10: Synthetic mix ratio results (mid-stage TR suite; `DSAI585_nanochat_turk.ipynb`).

Synthetic mix	XNLI-TR (%)	Belebele-TR (%)	PLU-NE (%)	Avg (%)
0%	33.33	27.33	26.41	29.02
25%	34.15	26.22	26.11	28.83
50%	33.33	24.67	25.65	27.88
75%	34.35	28.56	26.56	29.82
100%	33.35	27.67	25.80	28.94

Table 11: Synthetic mix ratio results on held-out synthetic EVAL suite (mid-stage; `DSAI585_nanochat_turk.ipynb`).

Synthetic mix	SYN-XNLI (%)	SYN-XCOPA (%)	SYN-Belebele (%)	SYN-PLU (%)	SYN-MMLU-TR (%)	Avg (%)
0%	35.14	51.11	16.67	21.43	20.00	28.87
25%	27.03	44.44	20.83	37.14	0.00	25.89
50%	27.03	55.56	35.42	22.86	60.00	40.17
75%	27.03	53.33	18.75	25.71	40.00	32.96
100%	27.03	51.11	20.83	37.14	0.00	27.22

These synthetic-suite results are from mid-training checkpoints; SFT-stage *synthetic-suite* results will be added once available.

Table 12: Synthetic mix ratio results (SFT-stage TR suite; `DSAI585_nanochat_turk.ipynb`).

Synthetic mix	XNLI-TR (%)	XCOPA-TR (%)	Belebele-TR (%)	Avg (%)
0%	34.77	50.00	27.33	37.37
25%	31.84	49.80	29.78	37.14
50%	35.74	50.00	29.33	38.36
75%	31.74	48.40	27.56	35.90
100%	33.50	46.80	26.00	35.43

6 Results

6.1 Main benchmark results

Table 13: CETVEL subset benchmark results (mid-stage, d10 mix runs from `DSAI585_nanochat_turk.ipynb`).

Model	XNLI-TR (%)	Belebele-TR (%)	PLU-NE (%)	Avg
Baseline nanochat-TR (∼133M)	33.33	27.33	26.41	29.02
SynAug nanochat-TR (∼133M)	33.35	27.67	25.80	28.94

We report absolute accuracies and deltas versus the baseline, plus a simple macro-average to summarize performance across tasks. Where possible, we

include confidence intervals and note any evaluation differences (prompt templates, context limits) that could affect comparability with external baselines.

6.2 Synthetic benchmark results

Table 14: Synthetic benchmark results (held-out, mid-stage; MCQ-only synthetic suite).

Model	Synthetic MCQ Acc (%)	Synthetic Instruction (loss / judge score)
Baseline nanochat-TR	28.87	n/a (MCQ-only)
SynAug nanochat-TR	27.22	n/a (MCQ-only)

These results are designed to stress generalization to unseen synthetic prompts and to validate that improvements are not limited to the training templates. Where available, we include judge score distributions to show quality variance rather than only mean scores.

6.3 Synthetic data verification results

Table 15: Synthetic data verification summary.

Split	#Candidates	#Accepted	Acceptance (%)	Key failure modes
Synthetic TRAIN	300	216	72.0	wrong MCQ label, morphology errors, float artifacts, low-clarity items
Synthetic EVAL	500	242	48.4	morphology errors, RC distractor ambiguity, MCQ label errors

We report acceptance rates and failure modes by stage (schema, language, judge), which helps interpret whether synthetic data quality is the main bottleneck.

6.4 Qualitative examples

We will include 3–6 paired examples comparing baseline vs SynAug outputs, covering instruction following, reasoning, and failure cases. Examples will be selected from held-out prompts and will include the exact prompt, decoding settings, and a brief qualitative analysis. We include a nanochat-92M chat excerpt (Fig. 1) as a qualitative example of failure modes. Additional paired examples comparing baseline vs SynAug outputs are deferred to future work.

en sevdiğin renk hangisi

en sevdiğiniz renk sarıdır.

yok ben seninkini sordum

Evet, üzgünüm. "En sevdiğim renk hangisi?"

yeşil

Bu ifade çok geneldir ve kırmızı, mor, sarı veya turuncu gibi farklı renkler içerebileceğini ima eder.

genel filan değil, bildiğin yeşil işte

Öncelikle, yeşil rengin bazı özelliklerini tanımlayayım. Yeşil, yeşil, yeşil, turuncu veya turuncu arasında ayrım yapma yeteneğidir. İnsanlar yeşil renginin bazı özelliklerini açıklayabilseler de, farklı tonlarını takdir edebilirler. Örneğin, yeşil bir renk, koyu kahverengi bir rengi tercih ederken, turuncu bir renk, koyu kırmızıyı tercih eder. Benzer şekilde, yeşil bir renk, yeşil bir renk, mavi veya turuncu veya turuncu olmak üzere farklı tonlarını ifade edebilir. Bu, yeşil rengin daha geniş bir renk yelpazesi için iyi bir seçim olabileceğini düşündürmektedir. Son olarak, yeşil bir renk, yeşil olduğu anlamına gelir, ancak bunu söylemenin daha az pratik veya daha basit bir yoludur. Genel olarak, daha geniş bir seçim yelpazesine sahip olmak, herkes için en iyi rengi belirlemenin bir yolu olduğundan, yeşil rengin bazı özellikleri hakkında fikir edinmenin harika bir yoludur.

neyse başka bir şey konuşalım, en sevdiğin hayvan hangisi

En sevdiğin hayvan, kırmızı, yeşil, sarı veya turuncu olabilir.

sevdiğin bir yemek var mı

Hayır, en sevdiğin hayvan bir çeşit, kırmızı veya turuncudur.

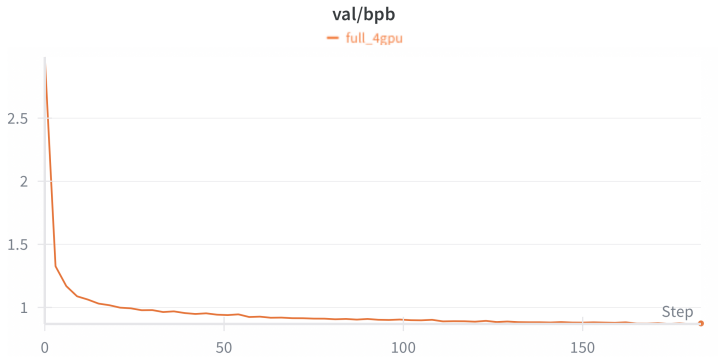
senin iq biraz düşük gibi kanka

muhtemelen hiç kanka yok.

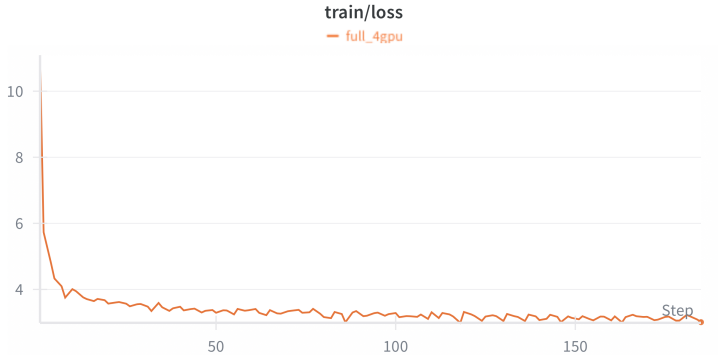
Figure 1: Back-to-back example outputs from a single nanochat-92M session.

6.5 Visualization (W&B training curves)

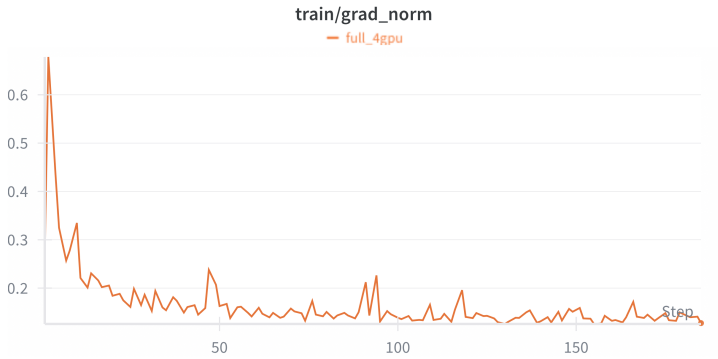
We log all training runs to Weights & Biases and include representative curves for validation BPB, training loss, and gradient norm. These plots serve as sanity checks for optimization stability and as evidence for convergence under the chosen batch sizes and learning-rate schedules. For the post-ablation main model (depth $d = 20$, 561M parameters, 11.2B tokens), training ran on 4×A100 GPUs for over 3 hours, consuming close to 3,000 compute hours on UHEM (Ulusal Yüksek Başarımlı Hesaplama Merkezi).



(a) Validation BPB.



(b) Training loss.



(c) Gradient norm.

Figure 2: W&B training curves for the main run (illustrative).

7 Discussion

7.1 What worked and why

The mid-stage synthetic mix sweep shows that moderate synthetic inclusion can help, but the best mix depends on the evaluation distribution. On the TR suite (CETVEL subset), the 75% synthetic mix achieves the highest macro average (29.82%), slightly above the 0% baseline (29.02%) and the 100% synthetic run (28.94%). This suggests that synthetic data can improve general Turkish performance when blended with original data, but a fully synthetic mix does not consistently outperform. On the held-out synthetic suite, the 50% mix is strongest (40.17%), while 0% and 100% are lower (28.87% and 27.22% respectively). This gap indicates a distributional trade-off: adding synthetic data helps generalize to synthetic-style benchmarks, but too much synthetic data can reduce performance on the original-distribution TR suite. Overall, the evidence points to a balanced mix (50–75%) as the most robust choice at mid-training, motivating our choice to continue with mixed ratios rather than fully synthetic training. In SFT-stage TR evaluations, the best macro average occurs at the 50% mix (38.36%), reinforcing the conclusion that a balanced synthetic/original mixture is preferable to fully synthetic or fully original data.

7.2 What did not work / limitations

- potential judge bias and model selection effects
- remaining risk of subtle leakage or template overfitting
- limited compute limiting ablation breadth
- mismatch between synthetic distributions and real Turkish usage
- limited coverage of long-context and multi-turn behaviors at 133M scale
- results reported here are mid-training only; SFT-stage trends may shift once instruction tuning is complete

7.3 Responsible practices

- leakage prevention checks and reporting
- safety filtering, PII avoidance
- transparent disclosure of any LLM assistance and judge settings
- dataset licensing and provenance tracking for all training sources
- clear separation of training and evaluation splits, including synthetic held-out sets

8 Conclusion

We adapted nanochat to Turkish and propose a verified synthetic data pipeline to improve small Turkish LLMs under limited compute. In mid-stage experiments, TRMorph segmentation consistently improves CETVEL-subset macro accuracy over baseline BPE in our tokenizer ablations, and synthetic-mix sweeps show competitive mid-training performance, with 75% synthetic mix yielding the best TR-suite macro average (29.82%). On the held-out synthetic EVAL suite, the 50% mix achieves

the highest macro accuracy (40.17%), indicating that synthetic data can improve generalization to synthetic benchmarks when balanced with original data. Beyond the core baseline vs SynAug comparison, the tokenizer ablation and efficiency reporting provide a practical roadmap for building Turkish SLMs that are deployable on constrained hardware. Future work includes scaling the verified synthetic pipeline, exploring longer-context settings, and testing on additional Turkish downstream tasks.

A Reproducibility

We provide links to exact commands, config files, random seeds, dataset versions, and hardware specifications (GPU type, memory, driver, PyTorch version), along with W&B run URLs and checkpoint hashes for the main experiments (see code archive and notebook).

Code and model downloads. All downloadable model artifacts are hosted at: https://drive.google.com/drive/folders/1A_pyzxgKhZKvygOD_uSPwEaF73wK7rnO?usp=share_link. All code snapshots used for the report are hosted at: https://drive.google.com/drive/folders/1DpFy8Lj33PIXTxqCMHx_Hv35vyxn8np1?usp=share_link. The live repository is maintained at: https://github.com/nurcunal/nanochat-turkish-synthetic.

Hardware and environment (from notebook logs). Experiments in `DSAI585_nanochat_turk.ipynb` ran on a single NVIDIA A100 GPU (Colab). Tokenizer vocabulary size is 65,536 and the default sequence length is 2048. The Turkish pretraining corpus is downloaded from `altaidevorg/fineweb2-hq-turkish` with a shard cap (e.g., 91 shards in the notebook run).

Synthetic data generation and judging. We generate and verify synthetic datasets using: `nanochat-master-turkish/scripts/synthetic_generate_openai.py` (teacher) and `nanochat-master-turkish/scripts/synthetic_judge_gemini.py` (judge). Both scripts accept/produce nanochat CustomJSON JSONL, so the resulting `accepted.jsonl` files can be plugged into training via the existing `tasks/customjson.py` loader. We will report the exact script arguments, the teacher/judge model identifiers, temperatures, and acceptance thresholds, and we will archive the generated `.meta.json` and `judge_report.json` artifacts for auditability.

B LLM-judge rubric and prompts

This appendix will list the judge prompt templates, scoring rubric, thresholds, and a small set of scored examples (accepted/rejected) to make the verification process auditable.

Rubric source of truth. The canonical judge prompt and acceptance policy are embedded in `nanochat-master-turkish/scripts/synthetic_judge_gemini.py`; the teacher constraints used for generation are embedded in `nanochat-master-turkish/scripts/synthetic_generate_openai.py`. We reproduce the rubric fields and thresholds in Sec. 4.3.2 and will include a small set of scored examples once synthetic generation is finalized.

References

[1] Joshua Ainslie et al. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[2] altaidevorg. Fineweb2-hq turkish (dataset card). https://huggingface.co/datasets/altaidevorg/fineweb2-hq-turkish, 2025. Accessed: 2026-01-16.

[3] Loubna Ben Allal et al. Smollm2: When smol goes big – data-centric training of a small language model. *arXiv preprint arXiv:2502.02737*, 2025.

[4] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 2022.

[6] EPFL. Fineweb2-hq tur_latn (dataset card). https://huggingface.co/datasets/epfml/FineWeb2-HQ, 2025. Subset: tur_Latn. Accessed: 2026-01-16.

[7] Jeremy Howard. Smollm training playbook. https://gist.github.com/jph00/3c97a2c6c5075c4e7b98faae634b033a, 2024. Accessed: 2026-01-17.

[8] Hugging Face. Smollm3: smol, multilingual, long-context reasoner. https://huggingface.co/blog/smollm3, 2025. Accessed: 2026-01-17.

[9] Andrej Karpathy. nanochat: The best chatgpt that $100 can buy. https://github.com/karpathy/nanochat, 2025. Accessed: 2026-01-16.

[10] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases. *arXiv preprint arXiv:2402.14905*, 2024.

[11] Meta. Meta llama 3. https://ai.meta.com/blog/meta-llama-3/, 2024. Accessed: 2026-01-17.

[12] Meta. Meta llama 3.2. https://ai.meta.com/blog/llama-3-2/, 2024. Accessed: 2026-01-17.

[13] OpenAI. Supervised fine-tuning guide. https://platform.openai.com/docs/guides/supervised-fine-tuning, 2025. Accessed: 2026-01-17.

[14] PyTorch Contributors. Scaled dot product attention (sdpa) — pytorch documentation. https://pytorch.org/docs/stable/generated/torch.nn.functional.scaled_dot_product_attention.html, 2024. Accessed: 2026-01-16.

[15] Qwen Team. Qwen model family. https://qwenlm.github.io/, 2025. Accessed: 2026-01-17.

[16] Qwen Team et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[17] Nureddin Cüneyd Ünal. nanochat-turk: nanochat adapted for turkish. https://github.com/nurcunal/nanochat-turk, 2026. Ongoing public project. Accessed: 2026-01-16.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[19] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Hannaneh Hajishirzi, et al. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.

[20] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zihao Wu, Yue Zhuang, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*, 2023.