

## Automated Java Method Naming Task Report

Li Zhen

**Experiments & Results:** I constructed the dataset using the SEART GitHub search engine. I queried for Java repositories with at least 100 commits, at least 10 contributors, a minimum of 10 stars, and excluded forks, with the search results sorted by number of commits in descending order. I downloaded the list and used a script to clone the top 500 repositories to my local machine. Using javalang, I parsed all Java files and extracted method declarations, stopping mining once I had about 2000k methods. I removed exact duplicates based on the method body, filtered out all constructor methods, filtered out any method whose name contained the string `test` (these were mostly auto-generated or project-specific names like `test\_0967`, `test\_0564`, etc.), and discarded methods longer than 256 tokens. In addition, to avoid noisy or potentially sensitive identifiers, I filtered out any method whose name or body contained substrings such as `accesssecret`, `access\_secret`, `accesskey`, `access\_key`, `secretkey`, `secret\_key`, `\_id`, `user\_id`, or `client\_id`. After these cleaning and filtering steps, I capped the dataset at 50k unique methods and then split it into 40k training methods and 10k test methods (80/20 split).

I first trained for one epoch with max sequence length 512, per-device batch size 1 and gradient accumulation 4 (effective batch size 4), using AdamW with learning rate  $1e-4$ ; this run had about 9.8k training steps, and a final training loss of about 2.32, yielding an exact-match accuracy of **44.6%** on the 10k-method held-out test set using greedy FIM inference with up to 8 generated tokens. After this initial run, I adjusted the training hyperparameters to learning rate  $5 \times 10^{-5}$ , weight decay 0.01, and warmup ratio 0.05, and retrained for one epoch with the same data and batch settings; this second run reported a final training loss of about 1.82 and achieved a higher exact-match accuracy of **48.84%** on the same 10k-method test set. Finally, I keep the same data and hyperparameters while increasing the number of epochs, I observed smaller but consistent gains: training for two epochs yielded an exact-match accuracy of **51.84%**, and training for three epochs further improved the exact-match accuracy to **52.34%** on the same 10k-method test set.

**Analysis:** Based on final improved model's prediction result which obtained an exact-match accuracy of 52.34 % (5234 correct, 4766 wrong). Accuracy is clearly length-sensitive: short names ( $\leq 10$  chars) reach 62.05 %, medium names (11–20) drop to 46.01 %, and long names ( $> 20$ ) to 32.65 %; the same pattern appears by subtoken count, with 62.69 % for 1-token names, 52.21 % for 2–3 tokens, and 36.83 % for names with more than 3 tokens. Subtoken coverage shows that in 99.23 % of all methods every subtoken of the true name appears somewhere in the method body (roughly 99 % for both correct and wrong cases), so the relevant lexical cues are almost always present in the code. For wrong predictions, the average string similarity of 0.386 indicates that many errors are near-misses rather than completely unrelated names. Overall, these metrics suggest the model is strong on short, simple names but struggles to compose longer, multi-token names even when all the necessary subtokens are visible in the method body. I also checked two additional aspects. First, I looked for cases where the only difference between the true and predicted name is capitalization; there are only 20 such examples, which is about 0.4% of all wrong predictions, so simple case-sensitivity is almost never the reason a name is counted as incorrect. Second, I analyzed the length of the method bodies: in my test set they range from 3 to 256 approximate code tokens (average 36.22), and using data-driven cut points at 12 and 32 tokens I grouped them into short ( $\leq 12$  tokens), medium (13–32 tokens) and long ( $> 32$  tokens) methods. The model is most accurate on short methods (about 60% correct), slightly worse on medium methods (about 53% correct), and clearly struggles more on long methods (about 43% correct), which suggests that larger and more complex bodies make precise method naming significantly harder even when the relevant words are present in the code.

Building on this analysis, I looked more closely at the high-similarity “near-miss” errors and saw four main challenges for the model. First, many failures involve arbitrary or project-specific identifiers (case numbers, suffix digits, or internal IDs) where the code does not really contain enough information to choose the exact index, for example `mcpatcherforge$redirectColor17` vs `mcpatcherforge$redirectColor1`. Second, some errors

reflect missing very fine-grained semantic distinctions, such as confusing which numeric or color variant should be chosen, which requires deep understanding of project conventions rather than just local code cues. Third, a few predictions introduce slightly odd or unexpected characters or variants (for instance extra underscores or unusual capitalization), showing that the model sometimes drifts toward locally plausible but stylistically inconsistent tokens. Fourth, for long composite names the model struggles to keep every segment stable, as in `extractNestedBootStrapJar` vs `extractNestedBootstrapJar` or `writeFloat8Vector` vs `writeFloat4Vector`, where it preserves most of the structure but silently alters one subpart (e.g., “BootStrap” vs “Bootstrap” or “8” vs “4”). Together, these patterns suggest that the remaining errors are concentrated in very detailed, convention-heavy aspects of naming, rather than in understanding the broad intent of the method.

Based on the “most off” wrong predictions with zero string similarity, many pairs have reasonably high semantic similarity (for example `register` vs `loadFormATS`, `process` vs `visit`, `capacity` vs `getKeysCount`, `load` vs `reset`), which means the model often stays in the right functional neighbourhood but confuses different lifecycle stages or roles such as loading versus resetting, processing versus visiting, or capacity versus key count. Several other pairs involve very short and generic names (`save` vs `load`, `choose` vs `get`, `add` vs `put`, `of` vs `build`), where the code context is too underspecified and the model defaults to another common verb with similar frequency, so any single choice looks arbitrary. There is also a smaller set of genuinely off cases where both string and semantic similarity are low (for example `fromTag` vs `init` or `init` vs `checkResources`), which suggests that, for some methods, the available lexical and structural cues in the body are not enough for the model to recover the developer’s naming intent. Overall, these “most off” errors highlight that residual mistakes mainly arise when the name is extremely short or overloaded, or when several semantically close but functionally distinct actions are plausible and the model chooses the wrong one.

As I also inspected the wrong predictions more closely, I compared how often subtokens from the true name versus “false-only” predicted subtokens appear in the method body and found that in 78.68% of wrong cases the true subtokens actually occur more frequently, only 15.93% favor the predicted-only subtokens, and 5.39% are ties, suggesting that the model’s mistakes are not simply driven by token frequency in the code but by stronger learned priors over naming patterns. I then analyzed token-set relationships and observed that almost none of the errors are just re-ordering of the same subtokens (0.04%), while 8.04% are “over-specified” (prediction is a superset of the true subtokens), 11.69% are “under-specified” (prediction is a subset of the true subtokens), and the remaining 80.23% are mixed or disjoint sets, indicating that most residual errors involve genuine content mismatches rather than trivial order changes, with a noticeable portion reflecting overly long or overly short but still related names.

I analyzed the subset of wrong predictions that a WordNet-based heuristic classified as “synonym-like”: quantitatively, about 14.02% of all wrong cases (668 out of 4 766) fall into this synonym-like bucket, 0% into an “ambiguous-like” bucket under the same criteria, and the remaining 85.98% into other wrong cases. Within the synonym-like subset I observed six recurring patterns: (i) same operation but slightly different object or scope, such as `getLoginUserInfo` predicted as `getLoginInfo` or `getRedisCacheInfo` predicted as `getCacheInfo`, where the model chooses a closely related but more general or different target entity; (ii) morphological or minor lexical variants, for example `getAssigns` predicted as `getAssignments`, `loginInfoList` predicted as `getLoginInfoList`, or `isVarArg` predicted as `isVarArgs`, which are almost interchangeable from a human perspective but still counted as wrong under exact-match; (iii) symmetric or closely related operations, such as `exitUnitToUnitInterval` predicted as `enterUnitToUnitInterval`, `onChildViewRemoved` predicted as `onChildViewAdded`, or `dropColumnFamily` predicted as `deleteColumnFamily`, where the model clearly understands the conceptual family but sometimes predicts the “twin” action instead of the reference one; (iv) type or role confusions and small qualifier changes, including cases like `getRelType` predicted as `getRelName`, `buildBasePosDeleteWriter` predicted as `buildPosDeleteWriter`, or `getShaderPath` predicted as `getShaderName`, where the underlying resource is the same

but the predicted name refers to a different attribute or a more or less specific variant; (v) genuinely semantic shifts that would be problematic in practice, such as `isPhoneDuplicated` predicted as `isEmailDuplicated`, `getSafeHoles` predicted as `getUnsafeHoles`, `getWarnsCount` predicted as `getErrorsCount`, or `getZ` predicted as `getY`, where the model mixes nearby but meaningfully different concepts (phone vs email, safe vs unsafe, warnings vs errors); and (vi) opaque, ID-like names (e.g., `aTaintCase0033` predicted as `aTaintCase0013`), where the model stays in the right family prefix but fails to guess the essentially arbitrary numeric or suffix component. Overall, these metrics and patterns show that the synonym-like bucket captures a non-trivial but still minority portion of the errors where the model’s prediction is not random noise but a semantically related name; within this subset there are two main groups, benign variants (morphology changes, extra “get”, plural vs singular, slightly wider or narrower scope) that would often be acceptable in real code, and genuinely harmful shifts (safe vs unsafe, phone vs email, `NetId` vs `UserId`, warnings vs errors) where the model has learned the right region of the naming space but still picks the wrong facet, suggesting that improving semantic precision remains an important open problem.

Finally, I compared the methods that were previously mispredicted by the 48%-accuracy model but correctly predicted by the 52%-accuracy model, and I observed several consistent improvement patterns from the model. In many cases the new model moves from generic but plausible names to more precise ones (for example, refining `getConfig` into `getApiConfig` or `create` into `createDumpDirectory`), showing better use of contextual cues in the body. It also corrects the polarity or role of operations, more reliably distinguishing “enter” versus “exit”, “create” versus “drop”, or add/remove-style actions based on control-flow and side effects. The improved model adheres more closely to API and framework naming conventions (for example Android callbacks, manager classes, or SQL-style identifiers), and produces cleaner boolean/state predicates by systematically preferring predicate forms like `isRequired` or `isNotEmpty` over bare nouns or verbs. I further see more accurate handling of singular versus plural and collection semantics, where the model now chooses names such as `listOptimizers` or `getServerCatalogs` when the code manipulates collections, indicating sharper discrimination among closely related alternatives rather than settling for a nearby but incorrect sibling name.

In conclusion, the model is clearly doing more than just copying the most frequent tokens from the method body: in the majority of wrong cases, the true subtokens actually appear more often in the code than the “false-only” predicted subtokens, and many errors involve different token sets rather than simple re-ordering or capitalization glitches. This means the model has learned some non-trivial semantic and stylistic priors over method naming (for example, preferring certain verbs or patterns in a given context), but it still often picks the wrong choice among several semantically plausible candidates, especially for very short or highly overloaded verbs and for long composite names with project-specific suffixes or IDs. In the future, we can improve this by adding semantic-aware objectives (for example, treating close synonyms and near-miss spellings as softer errors), incorporating richer project-level context into the input, and using a reranking or calibration step that prefers names whose subtokens align more consistently with the method’s behavior.

### Screenshots:

Dataset:

```
Total collected methods (before dedup): 200035
Reached max_unique = 50000, stopping dedup.
Total unique methods collected (capped): 50000
Train examples: 40000, Test examples: 10000
```

First training result:

```
{'loss': 1.4922, 'grad_norm': 20.7948646541816, 'learning_rate': 7.182272727272727e-08, 'epoch': 1.0}
{'train_runtime': 43428.8421, 'train_samples_per_second': 0.908, 'train_steps_per_second': 0.227, 'train_loss': 2.3181177932140113, 'epoch': 1.0}
100%|
Training done.
Saved fine-tuned model + tokenizer to qwen_methodname_finetune1
```

Second training result:

```
{'loss': 1.12658, 'grad_norm': 28.3585558482888, 'learning_rate': 5.35845829525848e-08, 'epoch': 1.0}
{'train_runtime': 45130.3599, 'train_samples_per_second': 0.874, 'train_steps_per_second': 0.218, 'train_loss': 1.8209629728218788, 'epoch': 1.0}
100%|
Training done.
Saved fine-tuned model + tokenizer to E:\qwen_methodname_finetune
```

Third training result:

```
{'loss': 0.6155, 'grad_norm': 42.33873748779297, 'learning_rate': 8.274610292547512e-08, 'epoch': 2.0}
{'train_runtime': 10856.1844, 'train_samples_per_second': 7.265, 'train_steps_per_second': 1.816, 'train_loss': 1.3578660262502804, 'epoch': 2.0}
100%|
Training done.
```

Fourth training result:

```
{'loss': 0.2688, 'grad_norm': 29.950210571289062, 'learning_rate': 7.829738771442808e-08, 'epoch': 3.0}
{'train_runtime': 16145.3366, 'train_samples_per_second': 7.327, 'train_steps_per_second': 1.832, 'train_loss': 1.071720242302812, 'epoch': 3.0}
100%|
Training done.
```

First test result:

```
W1117 01:18:25.378000 13004 .venv\Lib\site-packages\torch\d
Loaded 10000 test examples.
The attention mask is not set and cannot be inferred from in
Exact-match accuracy on test set: 44.62% (4462/10000)
E:\(qwen) PS D:\methodname_prediction>
```

Second test result:

```
Loaded 10000 test examples.
The attention mask is not set and cannot be inferred from
Exact-match accuracy on test set: 48.84% (4884/10000)
Saved all predictions to predictions2.jsonl
```

Third test result:

```
The attention mask is not set and cannot be inferred from :
Exact-match accuracy on test set: 51.84% (5184/10000)
Saved all predictions to predictions3.jsonl
```

Fourth test result:

```
(venv) PS D:\methodname_prediction> python evaluate
Using device: cuda
`torch_dtype` is deprecated! Use `dtype` instead!
Loaded 10000 test examples.
The attention mask is not set and cannot be inferred from
Exact-match accuracy on test set: 52.34% (5234/10000)
Saved all predictions to predictions4.jsonl
```

## Analysis of the final model’s results:

```
Total evaluated examples: 10000
Correct: 5234, Wrong: 4766
Accuracy: 52.34%

Accuracy by true method name length:
short (<=10 chars): 62.05% (4717 examples)
medium (11-20 chars): 46.01% (4358 examples)
long (>20 chars): 32.65% (925 examples)

Method body length statistics (in approximate code tokens):
min: 3, max: 256, average: 36.22, cut points: <= 12, 13-32, > 32

Accuracy by method body length (data-driven buckets):
short bodies (<= 12 tokens): 60.11% (2152 correct, 1428 wrong, 3580 examples)
medium bodies (13-32 tokens): 53.04% (1665 correct, 1474 wrong, 3139 examples)
long bodies (> 32 tokens): 43.19% (1417 correct, 1864 wrong, 3281 examples)

Accuracy by number of subtokens in true name:
1 token: 62.69% (2206 examples)
2-3 tokens: 52.21% (6374 examples)
>3 tokens: 36.83% (1420 examples)

Subtoken coverage of TRUE name in method body (all examples):
all subtokens appear: 9923 (99.23%)
some subtokens appear: 0 (0.00%)
no subtokens appear: 77 (0.77%)

For CORRECT predictions:
total: 5234
all subtokens appear: 5182 (99.01%)
some subtokens appear: 0 (0.00%)
no subtokens appear: 52 (0.99%)

For WRONG predictions:
total: 4766
all subtokens appear: 4741 (99.48%)
some subtokens appear: 0 (0.00%)
no subtokens appear: 25 (0.52%)

Case-only wrong predictions (capitalization differences only):
20 (0.42% of all wrong predictions)

Average similarity for wrong predictions (0-1): 0.386

Top 5 near-miss wrong predictions (high similarity):
-----
True: mcpatcherforgeriesredirectColor17 | Pred: mcpatcherforgeriesredirectColor1 | sim = 0.967
True: getMethodSignatureHashList | Pred: getMethodsSignatureHashList | sim = 0.963
True: redirectGrassSideOverLay4 | Pred: redirectGrassSideOverLay1 | sim = 0.960
True: extractNestedBootstrapJar | Pred: extractNestedBootstrapJar | sim = 0.960
True: redirectGrassSideOverLay3 | Pred: redirectGrassSideOverLay1 | sim = 0.960
-----
```

```
Top 5 most off wrong predictions (low similarity):
-----
True: register | Pred: loadFormATS | string_sim = 0.000 | semantic_sim = 0.778
True: process | Pred: visit | string_sim = 0.000 | semantic_sim = 0.714
True: capacity | Pred: getKeysCount | string_sim = 0.000 | semantic_sim = 0.737
True: save | Pred: load | string_sim = 0.000 | semantic_sim = 0.400
True: fromTag | Pred: init | string_sim = 0.000 | semantic_sim = 0.000
-----

Semantic-style breakdown for WRONG predictions (WordNet, symmetric subtokens):
synonym-like (high similarity, overlap, and some WordNet synonym subtokens): 668 (14.02%)
ambiguous-like (high similarity and overlap, but no synonym pair): 0 (0.00%)
other wrong cases: 4098 (85.98%)

Example synonym-like WRONG predictions (WordNet):
-----
True: handleBindException | Pred: handleException
True: relProvider | Pred: getRelProvider
True: generateWithVarargsKeys | Pred: generateWithSingleVarargsKey
True: removePhilist | Pred: removePhiInsns
True: getLoginUserInfo | Pred: getLoginUser
-----

For WRONG predictions: comparison of subtoken occurrences in method body
more FALSE-only subtokens than TRUE subtokens: 759 (15.93%)
more TRUE subtokens than FALSE-only subtokens: 3750 (78.68%)
equal or both zero: 257 (5.39%)

For WRONG predictions: token order and subset/superset patterns
same tokens but different order: 2 (0.04%)
TRUE tokens subset of PRED tokens (over-specified prediction): 383 (8.04%)
PRED tokens subset of TRUE tokens (under-specified prediction): 557 (11.69%)
mixed / disjoint token sets: 3824 (80.23%)

Totally wrong predictions (no overlapping subtokens and no WordNet synonym pair):
count: 1369 (28.72% of all wrong predictions)

Example totally wrong predictions:
-----
True: init | Pred: unbind | string_sim = 0.333 | semantic_sim = 0.000
True: readInt32Array | Pred: init | string_sim = 0.071 | semantic_sim = 0.000
True: build | Pred: toLinkConf | string_sim = 0.091 | semantic_sim = 0.000
True: clone | Pred: loadTableLoader | string_sim = 0.200 | semantic_sim = 0.667
True: updateDeps | Pred: addLoopIn anonymous inline | string_sim = 0.115 | semantic_sim = 0.429
-----
```