Django-Specific Code Smells: Catalog, Prevalence, and Automated Detection

Leopoldo Teixeira
Centro de Informática, Universidade Federal de Pernambuco
Recife, PE, Brazil
lmt@cin.ufpe.br

Eduardo Luiz Silva
Centro de Informática, Universidade Federal de Pernambuco
Recife, PE, Brazil
els6@cin.ufpe.br

Abstract

Django is a widely used Python web framework for database-backed application development. As with any actively used framework, Django projects can accumulate *code smells*: patterns that do not break functionality but signal design or implementation choices likely to hurt maintainability, readability, or performance. Existing Django-aware static analysis tools primarily target schema- and configuration-level issues, leaving performance-oriented and behavioral smells largely unaddressed. This paper investigates three research questions concerning such smells: their provenance in a catalog derived from Django-focused grey literature, their prevalence in real-world projects, and the reliability of their automated detection. To answer these questions, we built a static analysis tool that detects five Django-specific smells spanning the Model, View, and Template layers, using Python's built-in `ast` module for checks within Python code and a lightweight regex-based tokenizer over HTML templates. We applied the tool to a sample of 33 public Django repositories of varying size, where it flagged 4157 occurrences across the five smells; a manual inspection of 50 flagged occurrences (44 of them genuine, 88% overall) indicates detection reliability is reasonable overall but uneven across smells.

Keywords

Code Smells, Static Analysis, Django, Software Quality

1 Introduction

Django is one of the most widely adopted web frameworks for Python, valued for the breadth of tooling it provides for rapid, database-backed application development [5]. As with any actively used framework, Django projects can accumulate *code smells*, that is, surface-level patterns in the source code that do not prevent a program from working but signal design or implementation choices likely to hurt maintainability, readability, or performance [7]. Such smells might be more frequent among developers who have not yet internalized the internals of the framework, and Django's feature surface carries a non-trivial learning curve, making tool support for their detection particularly valuable for development.

Existing Django-aware static analysis tools, however, concentrate on schema- and configuration-level issues rather than behavioral or performance-oriented ones, such as views that accumulate excessive responsibility or database object creation inside loops (Section 4 details these). This paper presents an open-source static analysis tool that detects five such Django-specific smells, and investigates three research questions: (RQ1, Catalog) what Django-specific code smells are reported in practitioner-oriented (grey) literature; (RQ2, Prevalence) to what extent these smells occur in real-world, open-source Django projects; and (RQ3, Precision) how reliably the tool's automated detection aligns with manual inspection of flagged occurrences.

2 Related Tools

A small number of static analyzers exist for Django, but each targets a narrower layer than the behavioral and performance-oriented smells covered here. The `django-lint` tool targets schema and configuration hygiene (e.g., nullable field declarations and missing `settings.py` options) rather than behavioral code, and it appears unmaintained, still depending on APIs from superseded PyLint/`astng` versions [12]. djLint is, by contrast, actively maintained and template-aware in the same sense as this paper's Complex Template Logic check, but targets a different concern: It enforces template syntax and formatting consistency, not semantic complexity or business logic that has leaked into the presentation layer [6]. `pylint-django` [17] and `prospector` [2] make general-purpose linters aware of Django's conventions (e.g., recognizing model and view base classes) but add no new Django-specific smell rules of their own.

Generic tools such as Pylint [14], Flake8 [16], Bandit [15], and Radon [20] operate purely at the level of general Python style, complexity, or security, with no awareness of the Django layer. On the academic side, prior work has cataloged code smells specific to the Model-View-Controller architecture generally [1], but without a Django-specific catalog or a released detection tool. None of these tools, academic or practitioner, targets behavioral or performance-oriented smells such as God Views or loop-based object creation, the gap this paper's tool is designed to fill.

3 Methodology

This section describes how the smell catalog (RQ1) and the evaluation repository sample (RQ2) were assembled.

Catalog construction. The five smells covered by the tool were identified through a review of Django-focused grey literature [9–11], including practitioner blog posts, official Django documentation guidance, and community discussion threads, since Django-specific idioms are predominantly documented in these practitioner-facing sources. We retrieved sources via general web search using boolean queries combining "django" with smell/anti-pattern and performance/maintainability terms together with software-engineering terms,[1] Stack Overflow and Reddit discussion threads, and the official Django documentation. A source was kept only if it

[1] e.g., ("django") AND (problem* OR *smell OR anti-pattern*) AND ("software engineering" OR "developer")

was reasonably current, discussed a concrete, Django-specific code-level problem (rather than a generic Python or web-development concern), and, where it made a technical claim (e.g., about query behavior), that claim was cross-checked against the official documentation before being accepted into the catalog. The review surfaced a broader set of recurring Django-specific concerns. The five reported here were selected because each was mentioned independently across multiple sources and was concretely operationalizable as a static, per-file or per-function check, unlike more contextual or cross-cutting concerns (e.g., inconsistent URL naming conventions) that resisted a precise, low-false-positive detection rule.

Repository sampling. We retrieved candidate repositories from the GitHub Search API, filtered to public, non-fork, actively maintained (pushed within the last two years) Python repositories with at least 20 stars and Django-related topics, descriptions, or `README` content. We then verified each candidate as a genuine Django project via the presence of a `manage.py` file and/or a dependency manifest listing Django, rather than relying on metadata alone. We classified verified repositories as small, medium, or large using a module/model/view-count scheme (small: ≤ 3 modules, ≤ 10 models, ≤ 10 views; medium: 4–10 modules, 11–30 models, 11–30 views; large: > 10 modules, > 30 models, or > 30 views; each repository classified by majority vote across the three metrics, and ties were broken toward the larger class). Verification and classification stopped once the sample reached a balanced spread across all three size classes at the target sample size, yielding 33 repositories (13 small, 7 medium, 13 large) out of a larger pool of candidates returned by the search API. Future work sampling Django repositories at a larger scale could draw on tools such as SEART-GHS [3].

Scan scope. Within each repository, the tool excludes paths that are not hand-written application source before applying any check: Django migrations (auto-generated, not authored logic), test code (any `tests`/`test` directory, plus `test_*.py`, `*_test.py`, `conftest.py`, `tests.py`, or `test.py` files), static/media assets, and vendored or third-party code (`node_modules`, `.venv` or `venv`, `site-packages`, `vendor`). Within the remaining files, the God View and Fat Model checks are further scoped to files that are plausibly actual Django views or models, named or pathed like Django's view/model-layer convention (`views.py`/`models.py`, or a `views`/`models` package), or containing a recognizable Django view/model signal (a `django.shortcuts`/`django.views` import or `request`-taking function for views; a class with a `Model`-shaped base for models); applying these heuristics without such scoping produced implausibly large counts in large codebases during an earlier evaluation pass. The other three checks (loop object creation, missing `prefetch_related` or `select_related`, complex template logic) are performed in a broader per-file scope.

4 The Tool

The tool combines two static analysis mechanisms, depending on the source layer being checked. For the *Model* and *View* layers, it relies on Python's built-in `ast` module [18], which converts Python source code into an Abstract Syntax Tree (AST). The tool reads each `.py` file, parses it into an AST, and runs pattern-matching checks against the tree to flag occurrences of four of the five smells. The `ast` module does not apply to the *Template* layer, which is HTML rather than Python, so the tool instead scans `.html` files with a lightweight regex-based tokenizer for the fifth smell. In both cases, each match is recorded with its smell type, file name, and affected line range. Figure 1 summarizes this pipeline.

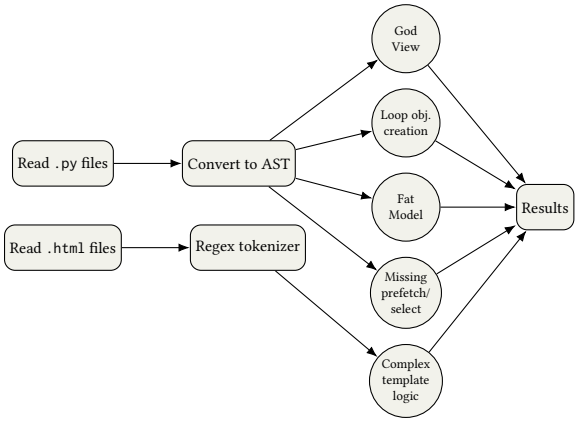

Figure 1: Execution pipeline of the tool. Python files are read and converted into an abstract syntax tree, which is passed to four parallel checks; HTML template files are read and scanned by a regex tokenizer for the complex template logic check. All five checks feed into a final results node.

4.1 God Views

Django views are recommended to remain small and focused [22]. A view that grows too complex tends to hurt both performance and readability. The tool flags a function that reaches 50 or more lines or nests control-flow blocks 3 or more levels deep, whichever threshold is reached first. Listing 1 shows a view accumulating several unrelated responsibilities, such as authentication, profile loading, form processing, and response rendering. Splitting such logic across multiple views or helpers, or adopting Class-Based Views [4], which structurally discourage this accumulation, are the recommended fixes.

```python
def user_profile_view(request):
    if request.method == 'POST':
        u_form = UserForm(request.POST)
        p_form = ProfileForm(request.POST, request.FILES)
        if u_form.is_valid() and p_form.is_valid():
            user = u_form.save()
            profile = p_form.save(commit=False)
            profile.user = user
            profile.save()
            user = authenticate(username=user.username,
    ↪ password=request.POST['password'])
            login(request, user)
            return redirect('profile_success')
    else:
        u_form = UserForm()
        p_form = ProfileForm()
    return render(request, 'user/profile.html', {
        'user_form': u_form,
        'profile_form': p_form
    })
```

Listing 1: Example of a God View in Django

4.2 Object Creation Inside Loops

Creating objects one at a time inside a loop is a common Django anti-pattern: it issues one database query per iteration instead of a single batched query [13]. Django's `bulk_create` (and, symmetrically, `bulk_update`) avoids this cost. Listing 2 performs one query per list element [21]. Listing 3 replaces it with a single `bulk_create` call, cutting queries to one regardless of batch size [21].

```
for index in range(number):
    User.objects.create(username=f"user-{index}")
```

Listing 2: Object creation inside a loop

```
users = [User(username=f"user-{index}") for index in
    ↪ range(number)]
User.objects.bulk_create(users)
```

Listing 3: Fix using bulk_create

4.3 Fat Models

Fat Models occur when business logic that should be decomposed into utility functions instead accumulates inside model classes, hurting single responsibility and testability [23]. The tool flags a `models.Model` subclass whose non-trivial methods (dunder methods and one-line `@property` getters excluded) amount to 4 or more, or whose combined lines across those methods reach 40 or more. Since `ast` cannot resolve imports or types, the check recognizes model classes via a name-based heuristic (a base class literally named `Model`) rather than true type resolution.

4.4 Missing prefetch_related and select_related

Missing use of `prefetch_related` and `select_related` when traversing related objects forces Django to issue one additional query per related lookup instead of resolving relationships in a single batched query [8]. The tool flags a related-object queryset access made off a `for` loop's target variable whenever the loop's originating queryset cannot be traced back, within the same function, to a `prefetch_related`/`select_related` call.

4.5 Complex Template Logic

Templates with complex logic, such as nested conditionals and business rules belonging in the view or model layer, are harder to read and maintain [19]. Since templates are HTML rather than Python, this smell uses a regex-based tokenizer flagging `{% if %}`/`{% for %}` tags nested more than 3 levels deep and custom filters absent from a small built-in allowlist, both signs that presentation-layer code carries logic that belongs elsewhere.

5 Availability and Prevalence

The tool is a self-contained command-line utility: given the root of a Django project, it walks the project's Python source files and HTML templates, applies the previously described checks, and reports each smell found together with the affected file and line range.

To answer RQ2 (to what extent these smells occur in real-world, open-source Django projects), we executed the tool over 33 public Django repositories of varying size, sampled following the procedure in Section 3 together with a size classification based on their number of modules, models, and views. Table 1 reports, for each smell and size class, the total number of occurrences flagged (after the scan-scope exclusions in Section 3) and the fraction of repositories in that class with at least one occurrence. The latter is a size-robust complement to the raw totals, since a single very large repository can otherwise dominate a sum across repositories of very different scale. God View is the most frequent smell overall (2584 occurrences) and appears in the large majority of repositories (28/33). Complex template logic is similarly widespread (26/33). Object creation inside loops is flagged in exactly two-thirds of repositories overall (22/33) but in *every* large repository in the sample (13/13). Missing prefetch/select_related and Fat Models are comparatively rarer and roughly comparable in frequency (17 and 16 of 33 repositories, respectively). These counts reflect the frequency of detected candidates rather than true prevalence: recall was not assessed in this study, so they are best read as a lower bound on how often each smell actually occurs, not a complete count.

Table 1: Occurrences flagged per smell, by repository size class (total occurrences; repositories with ≥ 1 occurrence / repositories in class)

Smell	Small	Medium	Large	Overall
God View	208 (9/13)	235 (7/7)	2141 (12/13)	2584 (28/33)
Object creation inside loops	26 (4/13)	47 (5/7)	231 (13/13)	304 (22/33)
Fat Model	11 (4/13)	11 (4/7)	77 (8/13)	99 (16/33)
Missing prefetch/select_related	18 (4/13)	11 (3/7)	347 (10/13)	376 (17/33)
Complex template logic	135 (8/13)	113 (7/7)	546 (11/13)	794 (26/33)

Occurrence density does not scale uniformly with repository size: God View occurrences average roughly 16 per repository among small projects but rise to approximately 165 per repository among large ones, an increase disproportionate to the roughly fourfold difference in repository count between these classes. This is consistent with larger codebases accumulating view-layer complexity faster than they accumulate views, though it may also partly reflect the coarser file-count-based size classification used here rather than a true per-view complexity effect (Section 7).

6 Manual Validation

To answer RQ3 (how reliably the tool's automated detection aligns with manual inspection of flagged occurrences), we manually inspected a stratified random sample of 50 flagged occurrences (10 per smell, fixed random seed for reproducibility) to assess whether each represented a genuine instance, without reference to the tool's internal detection logic during inspection. Table 2 reports the resulting precision per smell and overall (44/50, 88%).

Missing prefetch/select_related had the lowest precision (7/10). All three false positives shared one cause: the check treats any `.get(...)` call off the loop variable as a queryset lookup, but `get` is also Python's generic dict-access method, and each false positive's loop variable was in fact a plain `dict` (e.g., a parsed JSON response), not a queryset. God View's two false positives had distinct causes: one was a short function whose nesting depth reached the threshold without the function accumulating genuine responsibility, and the other was a nested class definition (a locally-defined form class), which the depth-counting heuristic treats the same as control-flow nesting even though it reflects structural, not logical, complexity. The Fat Model false positive was a model whose flagged methods

Table 2: Manual validation: genuine occurrences / inspected, per smell

Smell	Genuine/Inspected	Precision
God View	8/10	80%
Object creation inside loops	10/10	100%
Fat Model	9/10	90%
Missing prefetch/select_related	7/10	70%
Complex template logic	10/10	100%
Overall	44/50	88%

were standard validation hooks which Django convention places on the model itself, unlike the ad-hoc business logic the smell targets. Object creation inside loops and complex template logic had no false positives in this sample.

These results indicate that the tool's flagged occurrences are, for four of the five smells, reliable enough to support the prevalence claims without extensive manual filtering; missing prefetch/select_related's lower precision suggests its output should be treated as a starting point for manual triage rather than a direct count, at least until the `dict.get()` confusion identified above is resolved.

7 Threats to Validity

Construct validity. Several checks rely on structural heuristics rather than true type resolution. For example, the Fat Model check identifies model classes by a literal `Model` base-class name rather than resolving imports, and thresholds for method count, aggregate lines, and template nesting depth were set by judgment. The manual validation in Section 6 surfaced two concrete instances of this: missing prefetch/select_related's `get`-as-queryset-lookup signal collides with plain `dict.get()`, and God View's nesting depth count does not distinguish control-flow nesting from structural nesting (e.g., a class defined inside a method). The file-level scoping anchoring the God View and Fat Model checks to plausible view/model files (Section 3) is itself a heuristic: a view or model outside those naming/content conventions would be missed, while a matching file without genuine view/model code could still be checked unnecessarily.

Internal validity. The smell catalog and detection heuristics were both defined and implemented by the same team, which may introduce confirmation bias in what counts as a valid instance; the manual validation in Section 6 was likewise performed by the authors rather than independent raters, with no inter-rater agreement measure to cross-check the judgment calls involved. The validation sample itself is also small, as this work reports on preliminary results. Further work is needed to ground precision claims for the highest-volume rules.

External validity. Repositories were drawn from public GitHub rather than production codebases, which may not represent typical industry usage; the sampling criteria in Section 3 introduce their own selection bias. Raw occurrence totals are also sensitive to codebase size within a size class, so a single very large repository can account for a disproportionate share of a class's total, which is why Table 1 reports the fraction of repositories with at least one occurrence alongside the totals.

Reliability. The complex-template-logic check uses a regex-based tokenizer rather than a proper template-language parser, and so was expected to be the least precise of the five. Manual validation instead found it and object creation inside loops to be the only two checks with no false positives in the sample (Table 2), while missing prefetch/select_related (an AST-based check) had the lowest precision. This confirms that the manual validation's quantitative check was necessary rather than a formality. A recall-oriented evaluation, that is, estimating how many genuine smell instances the tool misses, was out of scope for this short paper and is left as future work.

8 Conclusion and Future Work

This paper presents a static analysis tool for detecting Django-specific code smells: five behavioral and performance-oriented patterns spanning the Model, View, and Template layers that existing Django-aware tools do not target (RQ1). Its application to a sample of 33 public repositories of varying size shows that these smells occur with non-trivial frequency in real-world Django projects (RQ2), and manual validation of 50 flagged occurrences indicates the tool is reasonably reliable overall (88% precision) but with per-smell precision ranging from 100% down to 70% (RQ3).

Future work includes replacing complex-template-logic's regex tokenizer with a proper template-lexer-based parser, IDE integration to surface warnings during development, and addressing the limitations identified in Section 7.

Artifact Availability

The tool source code and replication dataset are available at https://github.com/STAR-RG/django-smells and archived on Zenodo at https://doi.org/10.5281/zenodo.21840280. The repository includes the tool's implementation, usage instructions, the analysis scripts, the evaluation corpus, the obtained results, and the validation sample with scoring. The artifact is released under the MIT license.

Acknowledgments

We used Claude (Anthropic), via Claude Code and claude.ai, to assist in developing the Python analysis scripts (including those generating this paper's LaTeX macros/tables), and in generating the `TikZ` source for Figure 1. This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence),[2], CNPq grant 408817/2024-0. Leopoldo is partially supported by CNPq grant 310405/2025-4.

References

[1] Maurício Aniche, Gabriele Bavota, Christoph Treude, Marco Aurélio Gerosa, and Arie van Deursen. 2018. Code Smells for Model-View-Controller Architectures. *Empirical Software Engineering* 23, 4 (2018), 2121–2157. doi:10.1007/s10664-017-9540-2

[2] Carl Crowder. 2024. Prospector: Python Static Analysis Tool. https://prospector.landscape.io/. Accessed: 2026-07-10.

[3] Ozren Dabić, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories (MSR 2021)*. IEEE, 560–566. doi:10.1109/MSR52588.2021.00073

[2] www.ines.org.br

[4] Django Software Foundation. 2024. Class-Based Views, Django Documentation (6.0). https://docs.djangoproject.com/en/6.0/topics/class-based-views/. Accessed: 2026-07-10.

[5] Django Software Foundation. 2024. Django Documentation (6.0). https://docs.djangoproject.com/en/6.0/. Accessed: 2026-07-10.

[6] djLint. 2024. djLint: HTML Template Linter and Formatter. https://github.com/djlint/djLint. Accessed: 2026-07-10.

[7] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.

[8] Goutom Roy. 2019. Django select_related and prefetch_related. https://betterprogramming.pub/django-select-related-and-prefetch-related-f23043fd635d. Accessed: 2026-07-10.

[9] Fernando Kamei, Gustavo Pinto, Igor Wiese, Márcio Ribeiro, and Sérgio Soares. 2021. What Evidence We Would Miss If We Do Not Use Grey Literature?. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 24:1–24:11. doi:10.1145/3475716.3475777

[10] Fernando Kamei, Igor Wiese, Crescencio Lima, Ivanilton Polato, Vilmar Nepomuceno, Waldemar Ferreira, Márcio Ribeiro, Carolline Pena, Bruno Cartaxo, Gustavo Pinto, and Sérgio Soares. 2021. Grey Literature in Software Engineering: A critical review. *Information and Software Technology* 138 (2021), 106609. doi:10.1016/J.INFSOF.2021.106609

[11] Fernando Kenji Kamei. 2022. *Understanding and Supporting the Decision-Making Whether to Use Grey Literature in Software Engineering Research.* Ph.D. Thesis. Universidade Federal de Pernambuco, Centro de Informática, Recife, Brazil.

[12] Lamb, Chris. 2010. django-lint. https://github.com/lamby/django-lint. Accessed: 2026-07-10.

[13] Manuel Matosevic. 2022. Using Django bulk_create and bulk_update. https://zerotobyte.com/using-django-bulk-create-and-bulk-update/. Accessed: 2026-07-10.

[14] pylint-dev. 2024. Pylint. https://pylint.readthedocs.io/. Accessed: 2026-07-10.

[15] Python Code Quality Authority. 2024. Bandit: A Security Linter for Python Code. https://bandit.readthedocs.io/. Accessed: 2026-07-10.

[16] Python Code Quality Authority. 2024. Flake8: Your Tool for Style Guide Enforcement. https://flake8.pycqa.org/. Accessed: 2026-07-10.

[17] Python Code Quality Authority. 2024. pylint-django. https://github.com/PyCQA/pylint-django. Accessed: 2026-07-10.

[18] Python Software Foundation. 2024. ast — Abstract Syntax Trees, Python Documentation. https://docs.python.org/3/library/ast.html. Accessed: 2026-07-10.

[19] Qodo AI. 2023. Django Templates: Best Practices for Web Development. https://www.qodo.ai/blog/django-templates-best-practices-for-web-development/. Accessed: 2026-07-10.

[20] Rondelli, Michele. 2024. Radon: Code Metrics for Python. https://radon.readthedocs.io/. Accessed: 2026-07-10.

[21] Sabchuk, Michel. 2021. Load Large Sets of Data into a Postgres Database Using Python/Django. https://michelts.medium.com/load-large-sets-of-data-into-a-postgres-database-using-python-django-7137a59582a2. Accessed: 2026-07-10.

[22] Sallie Sorenson. 2022. 10 Django Views Best Practices. https://web.archive.org/web/20240803102205/https://climbtheladder.com/10-django-views-best-practices/. Accessed: 2026-07-10.

[23] Jair Vercosa. 2020. Django Model Guideline. https://jairvercosa.medium.com/django-model-guideline-d48a96c9b38c. Accessed: 2026-07-10.