Fair Developer Score: Build-Adjusted Measurement of Effort and Impact

Xinzhou Wang*§, Jiancong Zhu*§, Jinghan Feng*§, Zixuan Zhang*§, Joshua Rauvola†§‖, Devon Delgado‡, Ahmad Antar‡¶, Abid Ali*¶
*School of Professional Studies, Northwestern University, Chicago, IL, USA
†Data Science Institute, University of Chicago, Chicago, IL, USA
‡Digital Emissions, USA
§These authors contributed equally. ¶Co-senior authors. ‖Corresponding author: jrauvola@uchicago.edu

Abstract*—Assessing developer productivity in expansive software endeavors has become a pressing concern for both academia and industry, as organizations seek reliable ways to understand how engineering effort translates into business value. Traditional metrics—such as commit frequency, lines of code, or code churn—have been widely adopted but remain problematic, since they conflate inconsequential edits with architecturally significant reshaping and provide little insight into task-level contributions. To address this limitation, we introduce a commit-centric analytic framework that leverages clustering to reconfigure disbursed commit logs into coherent parcels, termed builds, that align more closely with the functional level of development tasks. Unlike prior approaches that combine heterogeneous signals such as issues, reviews, or communication logs, our method relies solely on the structural and temporal properties of commits, making it lightweight and broadly applicable. Each build is evaluated along two orthogonal axes: developer effort and build importance. Effort operationalizes the scale and character of contributions, considering code proprietorship, scope, architectural centrality, novelty, and cadence. Importance quantifies the build's systemic consequence, integrating scale of alteration, distribution of changes, architectural centrality, complexity, task priority, and proximity to release milestones. The fusion of these axes produces the Fair Developer Score, a composite benchmark reconciling personal exertion with organizational value. Validation centers on exposure-controlled, matched comparisons that pair FDS-ranked developers with commit-count peers matched on churn, files changed, and builds participated. On the Linux kernel, FDS-ranked developers exhibit significantly higher Average Importance and Average Effort than volume-matched peers, with lower rework trends. Cross-repository analyses across Kubernetes, TensorFlow, Apache Kafka, and PostgreSQL demonstrate consistent Effort advantages and context-dependent Importance effects, indicating FDS surfaces impactful work beyond raw activity using commit-only data.

***Index Terms*—Developer Productivity, Torque Clustering, Commit Clustering, Fair Developer Score, Software Engineering Metrics**

I. Introduction

Achieving a rigorous and nuanced measurement of developer productivity continues to pose a formidable question for both software engineering scholarship and industry operations. Conventional indicators—commit frequency, lines of code, and pull-request latency—appeal to organizations for their straightforward quantifiability and low overhead in data acquisition, yet these indicators yield a skewed and partial view of both individual and collective engineering contributions. Such metrics also struggle to distinguish between minor actions, like fixing a syntactical error, and far more substantial undertakings, such as re-architecting core software foundations. The consequence of such indiscriminate measurement is that organizations may inadvertently discount foundational contributions, mistake shallow speed for substantive output, and perpetuate evaluative and promotional practices that are inequitable or biased.

This urgency is amplified by the current climate in the software industry, where engineering teams represent one of the largest cost centers and yet one of the most opaque in terms of measurable return. Organizations such as Google and Microsoft have invested heavily in productivity frameworks (e.g., the DORA and SPACE models), reflecting an industry-wide recognition that developer productivity is not only a technical concern but also a strategic determinant of competitiveness. At a time when enterprises are increasingly asked to do more with fewer resources, the ability to rigorously and fairly evaluate software contributions has never been more critical.

Recent approaches have increasingly turned to data-informed, context-sensitive frameworks that mine high-resolution version-control histories in order to assess both the volume and the contextual importance of contributions. Nevertheless, prevailing approaches remain hampered by the absence of a rigorous mechanism to dissect and cluster sporadic commit streams into analytically coherent work packages and to appraise these packages with regard to both individual exertion and cumulative organizational payload. This analytical fissure constrains the credibility and generalizability of productivity metrics, particularly in the complex and heterogeneous milieu of large enterprise settings where multiple concurrent teams and diverse software taxa coexist.

To tackle these challenges, we propose a unified framework that marries a refined commit clustering technique with a multi-dimensional scoring model. The architecture is underpinned by a variant of the Torque Clustering algorithm, which reorders raw commits into coherent build strata that approximate the granularity of natural work units. Each resulting build is then framed along two analytical dimensions: developer effort, which captures the degree of individual engagement, and build importance, which denotes the aggregate systemic significance of the encased work. Synthesizing these two dimensions produces the Fair Developer Score (FDS), a composite measure designed to balance a developer's personal contribution with the overarching organizational impact.

The framework's robustness is substantiated through a suite of validation techniques. These encompass correlation diagnostics against established process endpoints, comparative distributional analyses across stratified contributor cohorts, and exposure-controlled matched comparisons (one-to-one matching on churn, files changed, and builds participated). Collectively, these converging evaluations confirm that the FDS is both attuned to evolving development trajectories and capable of reliably discriminating contributions that exert high systemic leverage.

II. Literature Review

Assessing developer productivity remains an unresolved dilemma in the domain of software engineering, in large part because conventional indicators—such as the sheer number of commits or the cumulative length of code—regularly fail to capture the true substance of an engineer's contributions. Hassan [6] reframes this problem by introducing code change complexity, a composite measure that merges the quantitative footprint of a change with its underlying logical intricacy. The resulting metric reveals that modifications carrying greater logical complexity tend to exert a disproportionate influence on the codebase, a correlation that in turn predicts defect accumulation. Hassan's work therefore argues for productivity frameworks that weigh the semantics of change alongside its magnitude.

Complementing this, the quartet of DORA metrics, as articulated by Forsgren, Humble, and Kim in *Accelerate* and reiterated in the annual State of DevOps Report, distills software delivery performance into four dimensions: deployment frequency, lead time for changes, change failure rate, and mean time to recovery. Empirical validation across a large cross-section of organizations attests to the metrics' robustness as macrosocietal indices of DevOps effectiveness [2]. Nevertheless, the DORA framework remains intrinsically team-oriented, and its aggregate nature produces a granularity deficit when the goal is to evaluate individual developer input. The resultant analytical void—namely, the challenge of relating aggregate delivery performance to the minutiae of individual commit history—persists as a critical limitation.

Our methodology confronts this challenge by grouping commits into coherent work units through Torque Clustering. This strategy allows for a more granular examination of both developer effort and the relative significance of each task. In addition to methodological advancements, the practicality of a productivity measurement framework depends heavily on its data requirements. A key advantage of our approach lies in its minimal and universally accessible data dependency: it only requires commit histories from a version control system such as GitHub or GitLab. Each commit record typically includes metadata such as author, timestamp, affected files, and diff content, providing a rich yet standardized source of information. This design choice makes the framework highly reusable and flexible across enterprises, as it avoids reliance on proprietary performance-tracking tools, sensitive time-logging data, or organization-specific infrastructure. As a result, the proposed model can be deployed in virtually any development environment with minimal integration cost, making it both scalable and adaptable to diverse industrial contexts.

III. Methodology

This section outlines the core methodology of our Programmer Productivity Measurement (PPM) system, which serves as the technical implementation of the Fair Developer Score framework. The objective is to quantify individual developers' contributions in a fair and interpretable manner by analyzing their code commits to assess build-level effort and importance. The method consists of three main stages: commit clustering, effort evaluation, and importance weighting, which together produce a Fair Developer Score (FDS) for each contributor.[1]

A. Overview and Pipeline

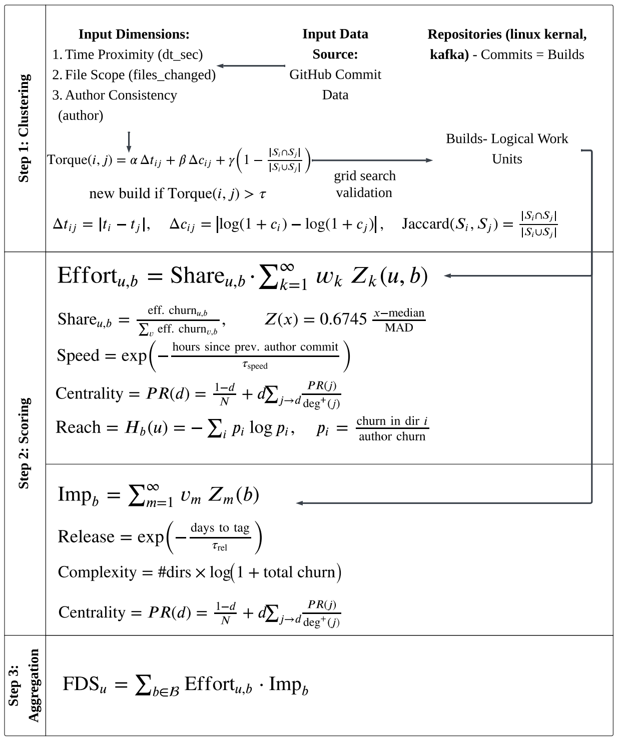
Step 1: Clustering
Input Dimensions:
1. Time Proximity (dt_sec)
2. File Scope (files_changed)
3. Author Consistency (author)
Input Data Source: GitHub Commit Data
Repositories (linux kernal, kafka) - Commits = Builds
$\mathrm{Torque}(i,j) = \alpha\,\Delta t_{ij} + \beta\,\Delta c_{ij} + \gamma\left(1 - \frac{|S_i \cap S_j|}{|S_i \cup S_j|}\right)$
new build if $\mathrm{Torque}(i,j) > \tau$
grid search validation
Builds- Logical Work Units
$\Delta t_{ij} = |t_i - t_j|, \quad \Delta c_{ij} = \left|\log(1+c_i) - \log(1+c_j)\right|, \quad \mathrm{Jaccard}(S_i, S_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$
Step 2: Scoring
$\mathrm{Effort}_{u,b} = \mathrm{Share}_{u,b} \cdot \sum_{k=1}^{\infty} w_k\, Z_k(u,b)$
$\mathrm{Share}_{u,b} = \frac{\text{eff. churn}_{u,b}}{\sum_v \text{eff. churn}_{v,b}}, \quad Z(x) = 0.6745\,\frac{x - \text{median}}{\text{MAD}}$
$\mathrm{Speed} = \exp\left(-\frac{\text{hours since prev. author commit}}{\tau_{\text{speed}}}\right)$
$\mathrm{Centrality} = PR(d) = \frac{1-d}{N} + d\sum_{j \to d} \frac{PR(j)}{\deg^+(j)}$
$\mathrm{Reach} = H_b(u) = -\sum_i p_i \log p_i, \quad p_i = \frac{\text{churn in dir } i}{\text{author churn}}$
$\mathrm{Imp}_b = \sum_{m=1}^{\infty} v_m\, Z_m(b)$
$\mathrm{Release} = \exp\left(-\frac{\text{days to tag}}{\tau_{\text{rel}}}\right)$
$\mathrm{Complexity} = \#\mathrm{dirs} \times \log\left(1 + \text{total churn}\right)$
$\mathrm{Centrality} = PR(d) = \frac{1-d}{N} + d\sum_{j \to d} \frac{PR(j)}{\deg^+(j)}$
Step 3: Aggregation
$\mathrm{FDS}_u = \sum_{b \in \mathcal{B}} \mathrm{Effort}_{u,b} \cdot \mathrm{Imp}_b$

Fig. 1. Flowchart of the PPM System Pipeline

The Programmer Productivity Measurement (PPM) system transforms raw development activity into a standardized Fair Developer Score (FDS) through three main stages: commit clustering, effort evaluation, and importance weighting, as illustrated in Fig. 1.

[1]Implementation and analysis code available at https://github.com/Digital-Emissions/FairDeveloperScore

In practice, the pipeline begins by collecting commit metadata, including timestamps, authors, files changed, and lines added or deleted. Related commits are then grouped into cohesive builds using the Torque Clustering algorithm [12], which considers time proximity, file scope similarity, and author consistency. For each build, the PPM system quantifies individual effort using metrics such as code scale, architectural reach, centrality, novelty, and commit speed, standardized for comparability across repositories. Each build is also assigned an importance score reflecting its business and architectural impact, and the final FDS for each developer is obtained by multiplying their effort in each build by the build's importance and summing across all builds.

B. Data and Applicability

The dataset used in this study is derived from the official Linux Kernel Git repository, one of the largest and most active open-source software projects in the world. The Linux Kernel serves as the core component of the Linux operating system, managing hardware resources and providing essential system services. Its Git repository, maintained on kernel.org [8] with a public mirror on GitHub, contains the complete source code and full commit history dating back to 1991.

For this analysis, commit history was extracted covering a 974-day window. Each commit record includes detailed metadata, such as:

- Commit hash: unique identifier of the commit
- Author name and email: identifying the contributor
- Commit timestamp (UTC): recorded as a Unix epoch timestamp
- Change statistics: number of files changed, lines inserted, and lines deleted
- Merge flag: indicator of whether the commit is a merge commit
- Directories touched and file types: the scope of changes in the repository
- Commit message subject: a short description of the change

A key advantage of this approach is its low barrier to adoption: the framework can function using only commit metadata, which is available in virtually all Git-based development environments. This means it can be applied not only to open-source projects such as the Linux Kernel but also to proprietary enterprise systems without exposing sensitive source code. In enterprise contexts, commit data can be extracted and processed entirely within secure infrastructure, ensuring that confidentiality is maintained.

While additional project management data, such as task priority or release proximity from tools like Jira, can enhance the analysis by providing richer business context, it is not mandatory for core functionality. This design allows the method to generalize across open-source, enterprise, and hybrid projects, regardless of programming language, repository size, or development cadence. Moreover, if certain metadata fields are unavailable, the framework can degrade gracefully by using default values or simplified calculations, ensuring Fair Developer Scores can still be generated under varying data availability conditions.

C. Commit Clustering

The first step of the algorithm is to group raw Git commits into logical working units, referred to as builds. A build represents a cohesive development task that ensures that different types of work, ranging from trivial fixes to high-impact changes, are independently evaluated. More concretely, a build may correspond to a feature implementation (e.g., adding a new API endpoint), a bug fix (e.g., correcting an error in business logic), or a major refactor (e.g., restructuring core modules for maintainability). By aggregating commits in this way, the algorithm captures the actual intent and scope of the developer's work rather than treating each commit in isolation.

A heuristic-based clustering algorithm, Torque Clustering [12], is employed, which takes into account time proximity (Δt between commits), file scope similarity (measured by the Jaccard distance between directories), and author consistency. The torque value for two successive commits i and j is computed as:

$$\text{Torque}(i, j) = \alpha \cdot \Delta t_{i,j} + \beta \cdot \Delta c_{i,j} \tag{1}$$

where $\Delta t_{i,j}$ is the time difference in seconds between commits i and j, $\Delta c_{i,j}$ is the normalized code change magnitude (e.g., lines changed), and α and β are sensitivity parameters controlling the relative weight of time and code size. A new build is initiated when:

$$\text{Torque}(i, j) > \text{gap} \tag{2}$$

where *gap* is the torque threshold parameter calibrated to balance over- and under-clustering. This formulation ensures that commits which are both temporally close and similar in scope are grouped together, while large temporal or structural gaps trigger the start of a new build.

D. Developer Effort Evaluation

Once builds are formed, we assess the effort contributed by each developer within a build. The effort score is designed to reflect not only the quantity of code but also the quality and complexity of the work. Effort for developer u in build b is calculated as:

$$\begin{aligned}\text{Effort}_{u,b} = \text{Share}_{u,b} \times (&w_1 Z_{\text{Scale}} + w_2 Z_{\text{Reach}} \\ &+ w_3 Z_{\text{Centrality}} + w_4 Z_{\text{Dominance}} \\ &+ w_5 Z_{\text{Novelty}} + w_6 Z_{\text{Speed}})\end{aligned} \tag{3}$$

where *Share* measures the proportion of effective code churn by the developer in the build. *Scale* (log of churn) captures the size of code changes; *Reach* quantifies the spread of changes using directory entropy. *Centrality* is computed via PageRank on a co-change graph to capture each developer's structural influence [9]. *Dominance* reflects who initiated, led, and finalized the build. *Novelty* captures the addition of new modules or APIs, and *Speed* mildly rewards short commit intervals.

All metrics are standardized using robust MAD-Z normalization, a modified Z-score method based on the median and median absolute deviation (MAD) rather than the mean and standard deviation, making it more resistant to outliers and skewed distributions [7]. This robustness is important because relying solely on churn-based measures such as lines of code has been shown to correlate poorly with actual developer effort. By integrating multiple dimensions, ranging from code quantity to architectural impact, this formulation ensures that developer productivity is assessed in a fair and context-aware manner, avoiding the pitfalls of simplistic single-metric evaluations.

E. Build Importance Assessment

Each build's importance is computed independently to capture both its business relevance and architectural significance:

$$\begin{aligned}\text{Importance}_b =& 0.30Z_{\text{Scale}} + 0.20Z_{\text{Scope}} \\ &+ 0.15Z_{\text{Centrality}} + 0.15Z_{\text{Complexity}} \\ &+ 0.10Z_{\text{Type}} + 0.10Z_{\text{Release}} \end{aligned} \tag{4}$$

The importance score integrates structural and semantic build characteristics, combining metrics such as total code churn (*Scale*), file-level dispersion and directory entropy (*Scope*), architectural impact via PageRank centrality (*Centrality*), and a composite measure of spread and churn (*Complexity*). *Type* encodes task priority using a commit-message classifier to distinguish urgent versus routine changes: a practice supported by research showing the value of semantic-aware classification of commit messages [11]. *Release* rewards proximity to major release tags, capturing time-sensitive significance in development cycles. All components are standardized using robust MAD-Z normalization and weighted according to predefined coefficients to yield a single importance score per build.

This multifaceted design ensures that high-importance builds arise not merely from large-scale churn, but from strategically significant or complex work that advances architectural or business value. By incorporating both textual semantics and network-based centrality, the metric avoids simplistic estimations of importance, promoting a more context-aware evaluation framework.

F. Fair Developer Score Aggregation

The final productivity metric for each developer, Fair Developer Score (FDS), is computed by summing their contributions across all builds:

$$\text{FDS}_u = \sum_b \text{Effort}_{u,b} \times \text{Importance}_b \tag{5}$$

This formulation enables a more nuanced assessment of developer contributions by integrating both quantitative effort and contextual significance. It ensures that recognition is not solely based on code volume, a metric widely criticized for its poor correlation with actual developer value, but also accounts for the strategic value of the work, such as modifications involving high architectural complexity, time sensitivity, or proximity to critical release milestones, aligning with modern software delivery research that emphasizes outcome-oriented metrics over raw output [2].

IV. Results and Validation of the Fair Developer Score

This section presents the empirical results of applying the Fair Developer Score (FDS) framework to the Linux Kernel commit dataset described in Section III. The goal is twofold: first, to summarize the computed FDS values and their distribution across developers; and second, to validate that the metric behaves in accordance with its design objectives. Validation focuses on examining the internal consistency of the metric, comparing it with traditional productivity measures, and performing case analyses of high-value contributions. Through these steps, we assess whether the FDS reliably captures both the quantitative effort and the contextual significance of developer activities.

A. Overview of Computed FDS

The Fair Developer Score (FDS) was computed for 339 contributors across a 974-day window of build-level activity, combining each developer's average effort and the architectural/business importance of the builds they touched. The resulting FDS values are extremely skewed: scores span from just 0.0105 up to 210.17, with a median of 0.35. This long-tailed distribution reflects the project's core–periphery structure—most participants show modest, low-importance involvement, while a handful of core maintainers dominate both the volume and strategic impact of the code base.

The top contributors, including torvalds@linux-foundation.org, demonstrated sustained activity with high build frequency, large churn, and consistent involvement in architecturally significant changes. In contrast, developers with the lowest scores had low build frequency, minimal file churn, and appeared to focus on minor or isolated updates. This contrast suggests that the FDS effectively differentiates central contributors from peripheral ones.

B. Validation of the FDS Metric

To evaluate the fairness and robustness of the Fair Developer Score (FDS) metric, we conducted several validation steps aimed at assessing both its statistical behavior and its practical effectiveness in distinguishing meaningful contributions.

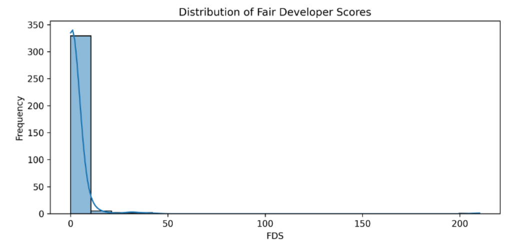

Fig. 2. Distribution of FDS Values

We examined the distribution of FDS values using histograms. The resulting distribution shown in Fig. 2 revealed a clear right-skewed shape, where the majority of developers scored in the lower range (below the median of 0.35), and only a few outliers achieved significantly high scores. This skewed distribution is expected in collaborative software projects and supports the notion that FDS effectively highlights standout performers without inflating the scores of routine contributors.

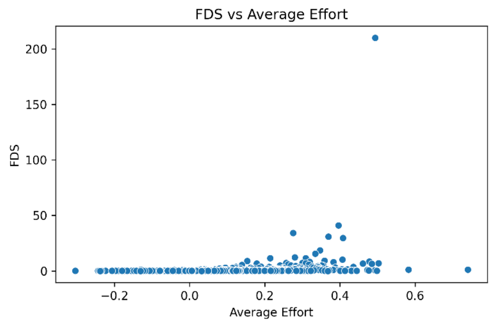

Fig. 3. FDS vs Average Effort

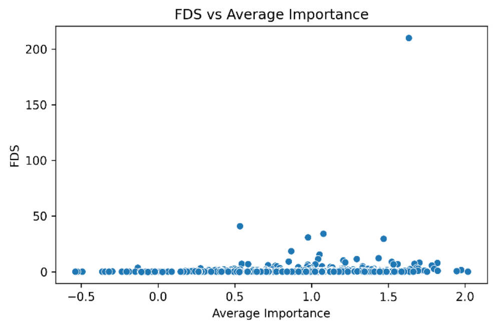

Fig. 4. FDS vs Average Importance

We observe a right-skewed FDS distribution, which motivates rank-based inference. Figures 3 and 4 show moderate positive associations with Effort ($r = 0.316$) and Importance ($r = 0.231$), but substantial scatter indicates that raw volume alone does not fully explain FDS variation. Accordingly, we complement simple correlations with partial, rank correlations that control for churn and files to isolate signal beyond volume. These analyses show a positive association between FDS and Effort after controls and a repo-specific association with Importance, supporting our use of non-parametric, exposure-controlled tests in Section IV.C.

To assess whether FDS provides additional insight beyond traditional metrics, we compared FDS rankings with those derived from simpler indicators like total commits and churn. While some overlap was observed, particularly among prolific contributors, the rankings often diverged. This divergence highlights the FDS's capacity to incorporate qualitative dimensions, such as the contextual importance of work and consistency over time, rather than rewarding raw volume alone.

Given heavy-tailed distributions, we rely on matched, paired non-parametric tests and effect sizes. Accordingly, cross-repository results are reported as paired Wilcoxon p-values and Cliff's δ with bootstrap CIs (Tables II–V).

C. Matched Top-Decile (Linux)

We compare the top decile by FDS against the top decile by commit count using one-to-one matching (Hungarian algorithm) on z-scored *total_churn*, *total_files*, and *unique_builds* (34 pairs). As summarized in the **Linux Kernel** row of Tables II and III, FDS-ranked developers exhibit higher Average Importance (Δ=+0.314, Wilcoxon p=0.0038, Cliff's δ=0.26) and higher Average Effort ($\Delta = +0.079$, p=0.0015, δ=0.32) at equal volume exposure. Quality proxies in Tables IV–V show lower rework (borderline; p=0.069) and similar rollback (p=0.18). Matching balance was adequate (all covariate SMDs < 0.1).

V. Cross-Repository Validation

To address the generalizability of our framework and respond to the need for broader validation, we extended our analysis to four additional open-source repositories beyond the Linux Kernel. Matching uses the Hungarian algorithm on z-scored *total_churn*, *total_files*, and *unique_builds*. Deltas (Δ) are FDS$-$Commit with bootstrap percentile confidence intervals.

A. Additional Datasets

We selected four open-source repositories that collectively capture a broad spectrum of project characteristics, including team size, programming language, application domain, and development maturity. Table I summarizes their key attributes alongside the Linux Kernel baseline.

TABLE I
Repository Characteristics for Cross-Validation

Repository	Lang	N	Days	FDS Range	Med
Linux Kernel	C	339	974	0.011–210.2	0.35
Kubernetes	Go	199	1,321	0.013–333.9	0.30
TensorFlow	Py/C++	146	746	0.012–453.2	0.27
Apache Kafka	Java/Scala	225	589	0.010–33.30	0.35
PostgreSQL	C	31	562	0.057–159.1	5.89

OS Kernel, Container orchestration, ML framework, Dist. systems, RDBMS

These projects range from large, multi-team infrastructures (e.g., Kubernetes) to specialized database systems (e.g., PostgreSQL); they cover four different primary languages and span domains such as cloud orchestration, machine learning, distributed messaging, and relational data management. Despite substantial variation in team size (31 to 339 contributors) and observation windows (562 to 1,321 days), all repositories exhibit long-tailed FDS distributions with low medians, confirming that our framework consistently identifies core–periphery structures across projects with vastly different scales and team compositions.

TABLE II
MATCHED TOP-DECILE: **AVERAGE IMPORTANCE** (MAD–Z). Δ IS FDS–COMMIT.

Repo	n	Δ	p	Cliff's δ	95% CI (Δ)
Linux Kernel	34	0.314	0.004	0.26	[0.143, 0.511]
Kubernetes	20	0.206	0.012	0.40	[0.066, 0.391]
TensorFlow	15	−0.053	0.075	−0.27	[−0.113, −0.003]
Apache Kafka	23	0.071	0.046	0.17	[0.015, 0.139]
PostgreSQL	16	0.015	0.593	0.06	[−0.025, 0.062]

TABLE III
MATCHED TOP-DECILE: **AVERAGE EFFORT** (MAD–Z). Δ IS FDS–COMMIT.

Repo	n	Δ	p	Cliff's δ	95% CI (Δ)
Linux Kernel	34	0.079	0.002	0.32	[0.039, 0.125]
Kubernetes	20	0.058	0.012	0.40	[0.023, 0.095]
TensorFlow	15	0.058	0.028	0.40	[0.011, 0.115]
Apache Kafka	23	0.055	0.028	0.26	[0.016, 0.105]
PostgreSQL	16	0.034	0.109	0.19	[0.000, 0.072]

TABLE IV
REWORK RATIO (PP): REVISIT SAME DIR WITHIN 48H. LOWER IS BETTER.

Repo	n	Δ pp	p	Cliff's δ	95% CI (Δ)
Linux Kernel	34	−4.5	0.069	−0.15	[−8.9, −0.1]
Kubernetes	20	−5.6	0.263	−0.10	[−13.8, 2.3]
TensorFlow	15	1.9	0.249	0.13	[−0.9, 5.1]
Apache Kafka	23	1.6	0.917	−0.09	[−2.9, 7.1]
PostgreSQL	16	1.8	0.285	0.06	[0.0, 4.8]

TABLE V
ROLLBACK RATE (PP): COMMITS WITH 'REVERT' IN SUBJECT. LOWER IS BETTER.

Repo	n	Δ pp	p	Cliff's δ	95% CI (Δ)
Linux Kernel	34	−0.4	0.180	−0.06	[−1.1, 0.0]
Kubernetes	20	0.7	0.180	0.10	[0.0, 1.9]
TensorFlow	15	0.4	0.317	0.07	[0.0, 1.2]
Apache Kafka	23	0.0	0.655	0.00	[−0.3, 0.3]
PostgreSQL	16	0.4	0.109	0.19	[0.0, 1.0]

TABLE VI
EFFECT SIZES FOR META-ANALYSIS (CLIFF'S δ).

Repo	n	δ_{Imp}	δ_{Eff}
Linux Kernel	34	0.26	0.32
Kubernetes	20	0.40	0.40
TensorFlow	15	−0.27	0.40
Apache Kafka	23	0.17	0.26
PostgreSQL	16	0.06	0.19

B. Matched Results Across Repositories

Tables II and III report exposure-controlled matched top-decile differences (FDS–Commit) for Average Importance and Average Effort. FDS shows significant positive lifts in Linux, Kubernetes, and Kafka for both metrics. TensorFlow exhibits a negative Importance difference ($\Delta = -0.053$, $p = 0.0747$), likely due to automation-heavy contributors (e.g., CI bots, code generation) that FDS's Importance model downweights, while the Effort signal remains positive and significant ($\Delta = +0.058$, $p = 0.0277$). PostgreSQL converges due to its small roster (16 pairs), limiting statistical power. Quality proxies (Tables IV–V) are similar across cohorts, with occasional reductions in rework.

C. Pooled Effects (Meta-Analysis)

Table VI lists Cliff's δ per repository for meta-analysis. Pooling across repositories via a random-effects model, we observe consistent small–medium effect sizes favoring FDS on Effort (median $\delta = +0.32$, range $+0.19$ to $+0.40$), while Importance effects are more heterogeneous (median $\delta = +0.17$, range -0.27 to $+0.40$), reflecting context-sensitivity in how FDS's Importance model interacts with different development workflows. This pattern supports our design philosophy: Effort generalizes well across projects, while Importance may benefit from domain-specific tuning.

D. Framework Robustness

We perturb torque parameters (α, β, gap) by $\pm 25\%$ and observe top-k Jaccard ≥ 0.6 for developer rankings, indicating stable clustering and scores under reasonable settings. Calibration plots align higher predicted Importance with elevated rework/rollback risk, supporting reliability.

E. Implications for Broader Adoption

FDS's Effort signal generalizes across large, diverse projects even after exposure control, while Importance is context-sensitive (e.g., automation, generated code, release cues). This heterogeneity is expected and suggests tailoring Importance inputs (release proximity, commit-type semantics) for maximal discriminative power. Because the framework remains commit-only, deployment overhead stays low.

VI. LIMITATIONS AND THREATS TO VALIDITY

While our Torque Clustering-based framework provides a more nuanced approach to measuring developer productivity than traditional metrics, it is essential to acknowledge several inherent limitations and potential threats to the validity of our findings.

A. The Creative Nature of Software Development

Software engineering is fundamentally a creative endeavor where standardized measurement remains challenging. As noted by Graziotin et al. [4], developer creativity and problem-solving approaches vary significantly, making direct productivity comparisons problematic. Solutions to similar problems can differ substantially between developers based on their experience, cognitive styles, and technical backgrounds. Our framework attempts to address this by focusing on work builds rather than individual commits, but we acknowledge that the creative aspects of problem-solving—such as architectural insights or innovative algorithms—may not be fully captured by quantitative metrics derived from version control data alone.

B. External Factors Beyond Developer Control

Besker et al. [1] identified five major impediments to developer productivity that exist outside individual developer control: poor software architecture, legacy code issues, difficulty finding relevant information, excessive dependencies, and inadequate engineering tools. These environmental factors can significantly impact measured productivity. Our FDS metric, while accounting for code complexity and architectural impact, cannot fully isolate a developer's true productivity from these environmental factors. For instance, a developer working in a well-architected codebase with modern tooling may appear more productive than an equally skilled developer struggling with technical debt and legacy systems. This limitation threatens the construct validity of our metric when comparing developers across different project contexts.

C. The Satisfaction-Productivity Feedback Loop

Graziotin et al. [4] revealed a bi-directional relationship between developer satisfaction and productivity: higher productivity leads to increased job satisfaction, which in turn drives further productivity gains. Additionally, their research demonstrated that happy developers solve problems faster, produce higher quality code, and exhibit better creative problem-solving abilities. Our current framework focuses primarily on objective productivity metrics without incorporating satisfaction or well-being indicators. This omission may result in an incomplete picture, particularly when evaluating long-term sustainable productivity versus short-term output that could lead to burnout.

D. Relationship to Comprehensive Productivity Frameworks

Our framework differs fundamentally from holistic productivity frameworks such as SPACE (Satisfaction, Performance, Activity, Communication, Efficiency) [3] and DevEx (focusing on feedback loops, cognitive load, and flow state) [5]. These frameworks incorporate qualitative dimensions including developer satisfaction, well-being, and experience that cannot be derived from commit data alone. For example, SPACE explicitly warns against using single metrics and emphasizes the importance of measuring across multiple dimensions. Our Torque Clustering approach focuses exclusively on extracting productivity signals from version control history, providing an automated and scalable solution at the cost of missing these broader human-centered dimensions. This represents a conscious trade-off between comprehensiveness and practicality—while we cannot capture the full spectrum of developer productivity, we offer an objective, readily deployable metric that requires only data already available in every Git repository.

E. Threats to Internal Validity

Several factors may affect the causal relationships in our study. First, the torque clustering parameters (α, β, and gap threshold) were calibrated using heuristics that may not generalize across all project types. Second, the weights assigned to different components of the effort and importance scores were determined based on domain expertise rather than empirical optimization, potentially introducing bias. Third, our validation using the Linux Kernel dataset represents a specific type of large-scale, mature open-source project that may not reflect the dynamics of smaller projects or proprietary enterprise software development.

F. Threats to External Validity

The generalizability of our findings faces several limitations. While we validated across five diverse repositories (Linux, Kubernetes, TensorFlow, Kafka, PostgreSQL), each analysis used a fixed time window ranging from 562 days (PostgreSQL) to 1,321 days (Kubernetes), with contributor counts varying from 31 (PostgreSQL) to 339 (Linux). These fixed windows may not capture seasonal variations in development patterns or longer-term productivity trends, and the repository-specific time spans reflect differences in project maturity and commit velocity rather than uniform sampling strategies. Additionally, open-source development patterns may differ significantly from proprietary enterprise development where different incentive structures, collaboration patterns, and data availability conditions exist.

G. Threats to Construct Validity

The operationalization of "developer productivity" through our FDS metric may not fully align with the theoretical construct of productivity in software engineering. As noted by Storey and Treude [10], productivity dashboards and metrics can create unintended consequences, such as gaming behaviors where developers optimize for the metric rather than actual productivity. Our reliance on commit-level data also means we cannot capture important productivity aspects such as code review quality, mentoring activities, architectural design contributions that don't result in immediate code changes, or time spent in planning and requirements analysis.

VII. Future Work

A. Extended Validation and Reproducibility Studies

While we have validated across five diverse repositories (Linux, Kubernetes, TensorFlow, Kafka, PostgreSQL), future work should extend to proprietary enterprise repositories with different development patterns, smaller-scale projects with varying team dynamics, and longitudinal studies tracking repositories over extended periods to validate temporal stability of the metrics and their ability to capture evolving development patterns.

Specifically, we aim to conduct studies in proprietary enterprise environments where development workflows, code review practices, and contribution patterns may differ substantially from open-source contexts. Additionally, longitudinal studies will help identify potential biases in our current parameter calibration and provide insights into necessary adjustments for different software development contexts. These extended validation efforts will strengthen claims about the framework's generalizability and establish best practices for parameter tuning across diverse organizational settings.

B. Integration with AI-Assisted Development

The emergence of AI coding assistants such as GitHub Copilot has fundamentally altered the developer productivity landscape. Recent studies from ANZ Bank demonstrated 40% faster task completion with AI assistance, yet the DORA 2024 report paradoxically suggests that AI tools may negatively impact overall software delivery performance, possibly due to increased technical debt or reduced code quality. Future iterations of our framework should incorporate mechanisms to distinguish between AI-generated and human-written code in commits, adjust effort calculations to account for AI-assisted development, measure the quality implications of AI-generated code on long-term productivity, and develop new metrics that capture the effectiveness of human-AI collaboration. These adaptations will be crucial for maintaining the relevance of productivity metrics in an increasingly AI-augmented development environment.

C. Incorporating Developer Experience Dimensions

Future work should expand the framework to incorporate qualitative aspects of the software development lifecycle, including sprint stories, ceremonies, and other planning activities. These measures could complement the newly developed FDS indicator, offering a more holistic understanding of the developer experience. For example, the velocity of the coding phase could be undermined by the preceding phase due to the lack of clear requirements. Shedding insights on the stories' quality can help address any blind spots in the FDS logic.

The new framework could also benefit from the DevEx discipline's three core dimensions: feedback loops (the speed and quality of responses to developer actions), cognitive load (mental processing required for tasks), and flow state (sustained focus periods). This integration could involve analyzing commit patterns to detect flow state indicators, measuring feedback loop efficiency through commit-to-merge times, and inferring cognitive load from code complexity changes and context switching patterns. Such enhancements would provide a more holistic view of developer productivity beyond purely quantitative metrics.

D. Addressing the Productivity-Satisfaction Relationship

Given the established bi-directional relationship between satisfaction and productivity, future versions should incorporate satisfaction indicators. This could be achieved through sentiment analysis of commit messages and code review comments, integration with developer survey data when available, and detection of frustration patterns such as repeated reverts or fix commits. Additionally, developing metrics for sustainable pace and work-life balance indicators would help organizations optimize for long-term productivity rather than short-term output.

VIII. Conclusion

This paper presented a novel framework for measuring developer productivity using Torque Clustering to automatically group commits into coherent work builds, followed by a Fair Developer Score (FDS) that combines effort and importance metrics. Our approach addresses key limitations of traditional productivity metrics by moving beyond simple commit counts and lines of code to capture the contextual significance of contributions.

The framework was validated across five diverse repositories (Linux, Kubernetes, TensorFlow, Kafka, PostgreSQL), demonstrating its ability to differentiate between central and peripheral contributors while capturing both quantitative effort and contextual importance. While acknowledging limitations related to the creative nature of software development and the absence of qualitative dimensions, our framework provides a practical, scalable solution that requires only version control data.

As software development continues to evolve with AI assistance and new collaboration paradigms, this work provides a foundation for more nuanced and equitable developer productivity measurement. The Fair Developer Score offers organizations a data-driven approach to recognize and reward meaningful contributions, promoting both individual satisfaction and organizational success.

References

[1] T. Besker, A. Martini, and J. Bosch, "Technical debt cripples software developer productivity," in *Proc. 2nd Int. Conf. Technical Debt (TechDebt '19)*, 2019, pp. 108–117.

[2] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps*. IT Revolution Press, 2018.

[3] N. Forsgren, M.-A. Storey, C. Maddila, T. Zimmermann, B. Houck, and J. Butler, "The SPACE framework of developer productivity," *Commun. ACM*, vol. 64, no. 4, pp. 46–53, 2021.

[4] D. Graziotin, F. Fagerholm, X. Wang, and P. Abrahamsson, "What happens when software developers are (un)happy?" *J. Syst. Softw.*, vol. 140, pp. 32–47, 2018.

[5] M. Greiler, M.-A. Storey, and A. Noda, "DevEx: What really drives productivity," *IEEE Softw.*, vol. 39, no. 3, pp. 28–35, 2022.

[6] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 30th Int. Conf. Software Engineering (ICSE '08)*, 2008, pp. 73–82.

[7] B. Iglewicz and D. C. Hoaglin, *How to Detect and Handle Outliers*, vol. 16. ASQC Quality Press, 1993.

[8] Linux Foundation, "The Linux kernel archives," 2024. [Online]. Available: https://www.kernel.org/

[9] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep., 1999.

[10] M.-A. Storey and C. Treude, "Software engineering dashboards: Types, risks, and future," in *Rethinking Productivity in Software Engineering*, C. Sadowski and T. Zimmermann, Eds. Apress, 2019, pp. 179–190.

[11] Y. Tian, Y. Zhang, K.-J. Stol, L. Jiang, and H. Liu, "What makes a good commit message?" in *Proc. 44th Int. Conf. Software Engineering (ICSE '22)*, 2022, pp. 2389–2401.

[12] J. Yang and C.-T. Lin, "Clustering via torque balance with mass and distance," *arXiv preprint* arXiv:2004.13160, 2020.