Pseudocode Beats Prose: Comparing Instruction Formats for LLM Rule Compliance Across Multiple Structured-Output Domains

Bulat Yapparov
aictrl.dev
United Kingdom
bulat@aictrl.dev

ABSTRACT

LLM-based coding agents increasingly rely on *skill files*—structured instruction documents that specify style rules, constraints, and conventions for generated artifacts. Although skill files are widely adopted (e.g., Claude Code SKILL.md, Cursor rules), no study has compared how their *format* affects compliance. We present a controlled experiment comparing three conditions—no skill (baseline), Markdown prose, and Python pseudocode—across four structured-output domains (Chart, Dockerfile, SQL, and Terraform), five models from three families (Claude, GLM, and Gemini), and three task complexities. In 737 scored runs evaluated against 12–15 automated binary rules per domain, we find: (1) skill files reduce failure rates by 4.0× versus the baseline (33.4% to 8.4%; Cliff's $\delta = 0.781$, *large*); (2) pseudocode format further reduces failure rates from 9.8% to 7.1%, a 28% relative reduction versus Markdown ($p = 0.003$, Cliff's $\delta = 0.141$); (3) the pseudocode advantage is strongest on Claude Opus 4.6 (5.0% vs. 9.0% failure rate, a 44% relative reduction), while Gemini 3.1 Pro achieves the lowest overall failure rate (4.1% with pseudocode); and (4) pseudocode tightens the failure-rate distribution (variance ratio 1.60; Levene's $F = 9.38$, $p = 0.002$), increasing the probability of meeting a 10% production-quality threshold by up to 22 percentage points. We release all data, skills, evaluators, and analysis scripts at https://github.com/aictrl-dev/skill-md-research.

1 INTRODUCTION

Large language model (LLM) agents that generate code, configuration, and structured artifacts are now widely deployed in development workflows [15]. To steer these agents, practitioners write *skill files*: instruction documents specifying style rules, structural constraints, and domain conventions. Claude Code uses `SKILL.md` files; Cursor uses `.cursor/rules`; Windsurf uses `.windsurfrules`. SkillsBench [7] found that curated skill files raise agent pass rates by 16.2 percentage points, but examined only skill *presence* versus absence, not skill *format*.

Today, most skill files are written as Markdown prose—natural language with headings, bullet lists, and code examples. An alternative is to express the same rules as Python pseudocode: dataclasses for schemas, enums for valid values, and typed function signatures with validation logic. Pseudocode prompting has shown 7–16 F1 point gains on NLP classification tasks [13], but its effect on structured-output generation in realistic agentic domains is unstudied.

Meanwhile, He et al. [5] found that prompt format (Markdown, JSON, YAML, plain text) can cause up to 40% performance variation, and that the optimal format differs across model families (IoU < 0.2). This raises a practical question: *does it matter how we write skill files?*

We address this question with a controlled factorial experiment:

- **3 conditions**: no skill (baseline), Markdown prose, Python pseudocode—encoding the same rules in each format.
- **4 domains**: Vega-Lite chart generation, Dockerfile generation, dbt-style SQL pipelines, and Terraform infrastructure-as-code—covering different output modalities (JSON, Dockerfile, SQL, HCL).
- **5 models** from 3 families: Claude Haiku 4.5 and Opus 4.6 (Anthropic); GLM-4.7 and GLM-5 (ZhipuAI); Gemini 3.1 Pro (Google).
- **3 task complexities** per domain (simple, medium, complex), with up to 5 repetitions per cell.

This yields 737 evaluated runs, each scored against 12–15 domain-specific binary rules by automated evaluators.

Contributions. (1) The first empirical comparison of instruction *format* for LLM skill files, extending prior work on skill *presence* [7] and pseudocode on NLP tasks [13]. (2) A multi-domain evaluation across four structured-output domains and three model families, testing generalization. (3) Evidence that pseudocode reduces output *variability*, not just the mean, increasing the probability of meeting quality thresholds. (4) Open release of all skill files, evaluators, data, and analysis scripts.[1]

2 RELATED WORK

Prompt Format Effects. He et al. [5] compared plain text, Markdown, JSON, and YAML across six benchmarks, finding up to 40% variation on code translation for GPT-3.5-turbo and more robustness for GPT-4, but no single universally optimal format. Sclar et al. [16] quantified sensitivity to spurious formatting changes in few-shot settings, reporting up to 76 accuracy points difference on LLaMA-2-13B. Errica et al. [4] proposed sensitivity and consistency metrics for prompt rephrasings. Ngweta et al. [14] introduced Mixture of Formats to improve robustness. Liu et al. [9] jointly optimized prompt content and format. None of these studies included

[1] https://github.com/aictrl-dev/skill-md-research

pseudocode as a format or tested on agentic structured-output tasks.

Pseudocode Prompting. Mishra et al. [13] created pseudocode prompts for 132 NLP tasks from Super-NaturalInstructions, finding 7–16 F1 point improvements on classification and 12–38% ROUGE-L gains using BLOOM and CodeGen. Ablations showed that code comments, docstrings, and structural cues all contributed. Kumar et al. [6] extended this to training-time augmentation with pseudocode. Chae et al. [1] used pseudocode as an intermediate reasoning representation. These works tested traditional NLP tasks, not agentic structured-output generation.

Agent Skills. SkillsBench [7] benchmarked 86 tasks across 11 domains with 7,308 trajectories, finding curated skills raise pass rates by 16.2pp, but self-generated skills provide no benefit. Xu and Yan [19] formalized agent skills as composable packages. Neither compared skill *formats*. Our work extends SkillsBench by asking whether *how* the skill is written matters, not just *whether* one is provided.

Code as Prompt. Madaan et al. [12] showed that language models of code are few-shot commonsense learners, and Singh et al. [17] used code-like specifications for robotic task planning. These suggest that LLMs trained on code may respond better to code-formatted instructions—a hypothesis our pseudocode condition tests. Our use of validation functions like `violations()` builds on the self-refinement loops proposed by Madaan et al. [11], but moves the validation logic from a multi-turn feedback loop into the zero-shot instruction itself.

Constrained Decoding and Domain-Specific Generation. Willard and Louf [18] enforce schema compliance via constrained decoding at inference time. Our pseudocode approach achieves high compliance through structural priors alone, without requiring specialized inference-time engines. In domain-specific generation, Liu et al. [8] survey Text-to-SQL methods in the LLM era, and Lyu et al. [10] address automatic Dockerfile generation—two of the four domains in our evaluation.

3 METHODOLOGY

3.1 Research Questions

RQ1 (Skill Efficacy) Does having a skill file improve rule compliance over a no-skill baseline?

RQ2 (Format Effect) Does pseudocode format outperform Markdown prose for rule compliance?

RQ3 (Generalization) Does the format effect generalize across domains and model families?

RQ4 (Efficiency) How do skill files affect token consumption and per-run cost?

RQ5 (Reliability) Does pseudocode format reduce output *variability*, yielding tighter failure-rate distributions?

Table 1: Instruction conditions. Line counts are averaged across the four domains.

Condition	Format	Lines	Description
No skill	None	0	Task data + generic prompt
Markdown	Prose	211	Headings, tables, bullets
Pseudocode	Python	274	Dataclasses, enums, typed fns

Table 2: Evaluation domains with rule counts and task descriptions.

Domain	Output	Rules	Tasks (S / M / C)
Chart	Vega-Lite	15	GDP / AI trends / Cloud
Dockerfile	Dockerfile	13	Java / Analytics / Rust
SQL (dbt)	SQL	12	Revenue / Subs. / Returns
Terraform	HCL	13	S3 / VPC+EC2 / ECS

3.2 Independent Variable: Instruction Format

We test three conditions (Table 1). The Markdown and pseudocode skills encode the same rules, thresholds, and examples. However, the formats are not purely cosmetic: pseudocode provides additional *structural priors*—typed fields, closed enums, and explicit validation functions—that have no direct Markdown analogue (Section 5). Both skill conditions prepend the skill document to the task prompt; the no-skill condition includes only the task data and a generic instruction.

The pseudocode format uses Python-like dataclasses to define schemas (`@dataclass` with typed fields), enums to constrain valid values (`class Color(Enum)`), and validation functions (`def violations() -> list[str]`) that return a list of rule violations. The Markdown format expresses the same rules as prose with tables, code examples in fenced blocks, and checklists.

3.3 Domains and Tasks

We evaluate across four structured-output domains (Table 2), selected for diversity in output modality and the availability of deterministic automated evaluation.

Each domain has three tasks at increasing complexity. Chart tasks range from a simple bar chart to a multi-series line chart with annotations. Dockerfile tasks range from a single-stage Java application to a multi-target Rust monorepo build. SQL tasks range from a revenue aggregation with six models to a return-rate analysis requiring deduplication and window functions. Terraform tasks range from an S3 bucket to a full ECS Fargate deployment with ALB, RDS, and IAM.

3.4 Models

We test five models from three families spanning multiple capability tiers (Table 3), enabling cross-family and cross-tier comparisons following the finding by He et al. [5] that format preferences often do not transfer across model families.

Table 3: Models used in the experiment.

Model	Family	Tier	Interface
Claude Haiku 4.5	Anthropic	Economy	Claude Code CLI
Claude Opus 4.6	Anthropic	Frontier	Claude Code CLI
GLM-4.7	ZhipuAI	Mid-tier	OpenCode CLI
GLM-5	ZhipuAI	Frontier	OpenCode CLI
Gemini 3.1 Pro	Google	Frontier	Gemini CLI

All models were invoked with default temperature settings to reflect ecologically valid conditions. For the four original models, the three original domains (Dockerfile, SQL, Terraform) have 5 repetitions per cell yielding 4 models × 3 conditions × 3 tasks × 5 reps = 180 runs each. Chart uses 3 repetitions (81 runs after excluding GLM-4.7, which was not tested in that domain), giving 629 runs for the four original models. Gemini 3.1 Pro was added subsequently across all four domains with 3 repetitions per cell (3 conditions × 3 tasks × 3 reps × 4 domains = 108 runs). In total, we have 737 scored runs across five models and four domains.

3.5 Evaluation Pipeline

Each domain has a dedicated automated evaluator (Python script) that:

(1) **Extracts** the structured output from raw LLM response text (handling multiple response formats: JSONL, fenced code blocks, plain text).
(2) **Validates** structural integrity (e.g., at least one `FROM` instruction for Dockerfile, at least one `resource` block for Terraform).
(3) **Checks** 12–15 binary rules specific to the domain (Table 4).

The primary metric is **failure rate**:

$$\text{failure\_rate} = 1 - \frac{\text{rules\_passed}}{\text{rules\_scored}}$$

where `rules_scored` excludes conditional rules that are not applicable to a given task (e.g., the Terraform *sensitive* rule is only scored when the task contains secret values). We adopt failure rate as our primary metric rather than pass rate because, in the context of agentic automation, failure rate directly represents the human-in-the-loop intervention cost. A reduction from 10% to 1% failure represents a 10-fold improvement in automation reliability, a delta that is more clearly captured by failure-rate ratios than by absolute gains in pass-rate percentage points.

3.6 Statistical Methods

We use non-parametric tests throughout, as rule compliance scores are bounded and often ceiling-compressed.

- **Mann-Whitney U** (one-tailed for directional hypotheses RQ1–RQ2; two-tailed for RQ3 cross-family comparisons).
- **Cliff's delta** [2] for effect size, using standard thresholds: $|\delta| < 0.147$ negligible, < 0.33 small, < 0.474 medium, ≥ 0.474 large.

Table 4: Rule categories by domain. Each rule yields a binary pass/fail per run. SQL rules 1–10 produce per-file pass rates across all model files.

Domain	N	Rule Categories
Chart	15	Color palette, accent colors, accessibility (no red+green), title, source citation, font, data labels, y-axis origin, spines, grid, units, annotations, legend vs. direct labels, dimensions
Dockerfile	13	Base image tags, security (USER, secrets), structure (multi-stage, WORKDIR), cache (deps-first), layers (combined RUN), apt practices, HEALTHCHECK, EXPOSE, LABEL, exec form CMD, no ADD
SQL (dbt)	12	Keywords uppercase, one clause/line, table aliases, column aliases, no SELECT *, comment headers, LEFT JOIN only, COALESCE unknown, ROW_NUMBER dedup, one CTE/file, Jinja ref(), layer naming
Terraform	13	snake_case naming, variable description/-type, outputs, tags, variable grouping, file structure, no hardcoded IDs, provider pinned, backend configured, sensitive marked, data sources, locals block

Table 5: RQ1: Skill files vs. no-skill baseline (737 runs pooled).

Condition	N	Mean FR	Cliff's δ	p
No skill	246	33.4%	0.781 (large)	< 0.001
Skill (both)	491	8.4%		

- **Bootstrap 95% confidence intervals** (10,000 resamples, percentile method, seed 42) [3] for mean failure rates by model × condition.
- **Levene's test** (median variant) for equality of variances between conditions (RQ5). We report the variance ratio $\sigma^2_{\text{MD}}/\sigma^2_{\text{PC}}$ and 90% highest-density intervals (HDI) of the raw failure-rate distributions.

We report results per domain and pooled across all four domains. For pooled analyses, failure rates are computed per run (normalized to each domain's rule count) and then aggregated.

4 RESULTS

4.1 RQ1: Skill Files Reduce Failure Rates

Pooling both skill conditions (Markdown and pseudocode) against the no-skill baseline across all four domains and five models, skill files produce a **large** reduction in failure rate (Table 5).

The effect is consistent across all four domains (Table 6 and Figure 1). SQL shows the highest baseline failure rate (51.2%),

Table 6: RQ1 per domain: failure rate by condition.

Domain	No skill	Markdown	Pseudocode
Chart	22.8%	4.9%	2.3%
Dockerfile	17.5%	4.1%	3.8%
SQL (dbt)	51.2%	15.3%	10.2%
Terraform	37.8%	12.9%	9.9%
Pooled	**33.4%**	**9.8%**	**7.1%**

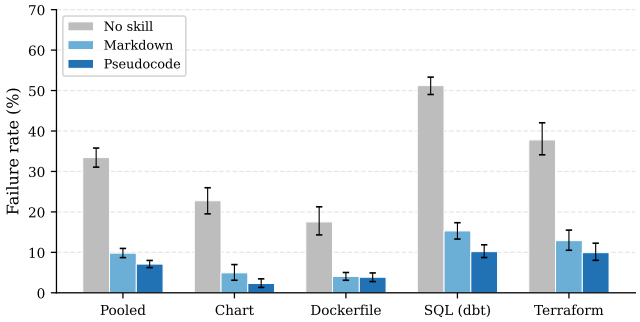

Figure 1: Failure rate by condition and domain with 95% bootstrap CIs. Skill files consistently reduce failure rates across all four domains, with pseudocode (dark blue) achieving the lowest rates.

while Dockerfile shows the strongest relative improvement (17.5% → 3.8%).

4.2 RQ2: Pseudocode Outperforms Markdown

Comparing the two skill formats directly, pseudocode reduces pooled failure rate from 9.8% (Markdown) to 7.1% (pseudocode), a 28% relative reduction. This is a statistically significant difference ($p = 0.003$, one-tailed), with a negligible-to-small effect size (Cliff's $\delta = 0.141$).

Per-domain results (Table 7) show that the effect is strongest in SQL ($\delta = 0.358$, medium). Chart shows a small effect ($\delta = 0.292$, $p = 0.009$) and Terraform shows a small effect ($\delta = 0.151$, $p = 0.032$). Dockerfile shows a negligible effect ($\delta = 0.034$), where both formats achieve near-ceiling compliance (<4.1% failure rate). A per-rule breakdown (Appendix C) reveals that the SQL advantage concentrates in two correlated rules where Markdown-guided models use a `SELECT *` shortcut in staging CTEs, while pseudocode models avoid it. In Terraform, the advantage comes from variable descriptions, types, and outputs—rules that map directly to pseudocode dataclass fields.

4.3 RQ3: Generalization Across Models and Families

Cross-domain consistency. All four domains show a positive pseudocode effect (Chart, Dockerfile, SQL, and Terraform). The mean Cliff's delta across domains is +0.209 (small positive). The binomial test for direction consistency (4/4 positive) yields $p = 0.12$.

Table 7: RQ2: Pseudocode vs. Markdown per domain. Positive δ favors pseudocode. $^*p < 0.05$.

Domain	MD FR	PC FR	δ	Mag.	p
Chart*	4.9%	2.3%	0.292	small	0.009
Dockerfile	4.1%	3.8%	0.034	negl.	0.347
SQL (dbt)*	15.3%	10.2%	0.358	medium	< 0.001
Terraform*	12.9%	9.9%	0.151	small	0.032
Pooled*	**9.8%**	**7.1%**	**0.141**		**0.003**

Table 8: RQ3: Cliff's delta (pseudocode vs. Markdown) by family and domain. Two-tailed tests. $^*p < 0.05$.

Family	Domain	δ	p	Mag.
Claude	Chart*	0.472	0.005	medium
	Dockerfile	0.100	0.435	negl.
	SQL*	0.982	< 0.001	large
	Terraform	0.252	0.053	small
GLM	Chart	0.099	0.750	negl.
	Dockerfile	−0.053	0.673	negl.
	SQL	0.093	0.532	negl.
	Terraform	0.033	0.798	negl.
Gemini	Chart	0.247	0.249	small
	Dockerfile	0.136	0.573	negl.
	SQL*	−0.704	0.007	large
	Terraform	0.222	0.169	small

Cross-family analysis. Table 8 and Figure 2 show the pseudocode effect separated by model family. For Claude models, the effect is statistically significant in SQL ($\delta = 0.982$, large) and Chart ($\delta = 0.472$, medium, $p = 0.005$), with Terraform trending positive ($\delta = 0.252$, $p = 0.053$), but not significant in Dockerfile. For GLM models, the effect is not statistically significant in any individual domain. For Gemini, the effect is positive in three of four domains, with small effects in Chart ($\delta = 0.247$) and Terraform ($\delta = 0.222$), though most do not reach statistical significance individually. Notably, Gemini shows a *negative* pseudocode effect in SQL ($\delta = -0.704$, large, $p = 0.007$), where pseudocode is significantly worse than Markdown—the only large negative effect in our data.

Frontier model analysis. The pseudocode advantage is most pronounced on Claude's frontier model (Table 9 and Figure 3). Claude Opus 4.6 shows a pooled failure rate of 5.0% (pseudocode) vs. 9.0% (Markdown), a 44% relative reduction. Gemini 3.1 Pro achieves the lowest failure rate of all models (4.1% pseudocode, 4.6% Markdown), with a small pseudocode advantage. GLM-5 shows 6.8% vs. 5.9%, with pseudocode slightly *underperforming* Markdown (+15% relative).

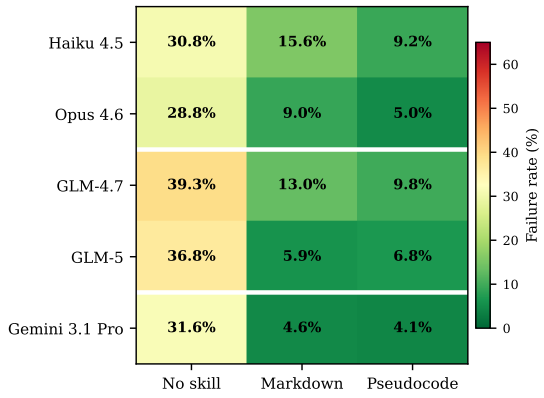

Figure 2: Failure rate (%) by model and condition. Darker green indicates lower failure rates. White lines separate Claude (top), GLM (middle), and Gemini (bottom) families.

Table 9: Failure rates for frontier models: Markdown vs. pseudocode.

Model	MD FR	PC FR	Δ	Rel. ↓
Claude Opus 4.6	9.0%	5.0%	+3.9pp	−44%
GLM-5	5.9%	6.8%	−0.9pp	+15%
Gemini 3.1 Pro	4.6%	4.1%	+0.4pp	−10%

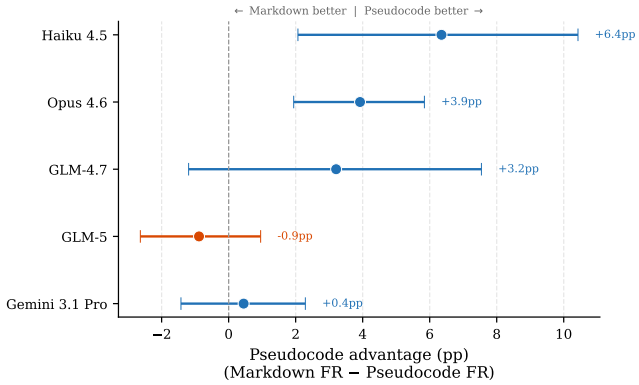

Figure 3: Pseudocode advantage per model (Markdown FR − Pseudocode FR) with 95% bootstrap CIs. Four of five models favor pseudocode, with Opus 4.6 showing the largest advantage; GLM-5's CI crosses zero.

4.4 RQ4: Token Efficiency

Skill files increase both input and output token consumption. Pseudocode prompts are 30.8% longer than no-skill prompts (5,597 vs. 4,279 mean input tokens), while Markdown adds 21.5% (5,198 tokens). Both formats also produce *longer* outputs: +46.2% for pseudocode and +40.3% for Markdown relative to the no-skill baseline (1,621 and 1,556 vs. 1,109 mean output tokens). Models generate more structured, multi-section output when guided by a skill file.

Comparing the two skill formats directly, pseudocode prompts are only 8% longer than Markdown and produce 4% more output tokens. For Claude models (where cost data is available), the mean cost per run is $0.061 (pseudocode) vs. $0.054 (Markdown) vs. $0.045 (no-skill). The pseudocode–Markdown cost premium of $0.007 per run buys a 28% relative reduction in failure rate (RQ2), a favorable tradeoff for production workloads where rework cost from non-compliant output far exceeds the token premium.

Table 10: RQ5: 90% HDI width of the failure-rate distribution and P(FR < 10%) per model. HDI narrowing indicates tighter distributions under pseudocode.

Model	HDI_{MD}	HDI_{PC}	Narrowing	ΔP(FR < 10%)
Haiku 4.5	38.5%	31.2%	+7.3pp (19%)	+22.2pp
Opus 4.6	17.3%	10.7%	+6.6pp (38%)	+20.9pp
GLM-4.7	30.8%	29.3%	+1.4pp (5%)	+8.9pp
GLM-5	15.4%	15.4%	+0.0pp (0%)	−1.8pp
Gemini 3.1 Pro	8.5%	9.8%	−1.2pp (−14%)	+2.8pp

4.5 RQ5: Pseudocode Reduces Output Variability

Beyond lowering mean failure rates (RQ2), pseudocode also *tightens* the failure-rate distribution. We assess this with three complementary measures: variance ratio via Levene's test, 90% highest-density interval (HDI) widths, and the probability of meeting a 10% production-quality threshold.

Pooled variance. Pooling across all models and domains, Markdown runs have a sample variance of 0.0086 compared to 0.0054 for pseudocode—a ratio of 1.60. Levene's test confirms unequal variances ($F = 9.38$, $p = 0.002$).

Per-model HDI narrowing. Table 10 shows the 90% HDI width of the raw failure-rate distribution per model. Pseudocode produces a narrower HDI for four of five models, with the largest narrowing on Opus 4.6 (17.3% → 10.7%, a 38% reduction). Gemini 3.1 Pro already has a very tight HDI under both formats (<10%), reflecting its consistently high compliance.

Production-quality threshold. The probability of a run achieving <10% failure rate increases under pseudocode for four of five models (Figure 4). The increase is largest for Haiku 4.5 (+22.2pp) and Opus 4.6 (+20.9pp). Pseudocode not only lowers the mean but concentrates runs in the low-defect region.

5 DISCUSSION

Why pseudocode works. We hypothesize that pseudocode's advantage stems from two mechanisms. First, *type constraints act as explicit contracts*: a dataclass field `sensitive: bool = True` is less ambiguous than the prose "mark sensitive variables with the sensitive flag." The LLM's code pre-training

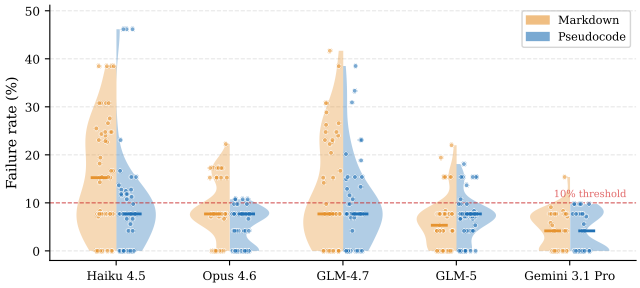

Figure 4: Failure-rate distributions by model and format (split violin). Left halves are Markdown; right halves are pseudocode. The dashed line marks the 10% production-quality threshold. Pseudocode distributions are tighter and more concentrated below the threshold.

makes it more likely to satisfy typed specifications. Second, *validation functions signal expected checks*: a `violations()` method that returns a list of failure conditions provides the model with an explicit checklist in a format it processes natively during code generation. Importantly, these mechanisms suggest that the pseudocode advantage is not merely a surface-level formatting effect. Typed dataclasses define *data structures*, enums create *closed token sets*, and validation functions specify *executable checks*—structural priors that are absent from equivalent prose. The benefit may therefore stem partly from providing a computationally-aligned representation that leverages the model's code pre-training, rather than from formatting alone. Ablation studies isolating the contribution of types, validation logic, and Python syntax would help disentangle these factors.

Relation to constrained decoding. Frameworks like Outlines [18] enforce structural compliance by constraining the token sampling process at inference time, guaranteeing schema adherence. Our pseudocode approach operates at the *prompt* level instead, achieving 92.9% pooled compliance through structural priors alone. The two techniques are complementary: constrained decoding guarantees syntactic validity, while pseudocode skills guide semantic compliance with domain rules.

Skill-induced complexity. The per-rule breakdown (Appendix C) reveals that skills can *hurt* compliance on individual rules. In SQL, both skill formats instruct multi-layer dbt pipelines with staging models. Most models take a shortcut—writing `SELECT * FROM source` in staging CTEs—that simultaneously violates two rules (clause-per-line and no-`SELECT *`). Without the skill, models write monolithic queries that avoid this pattern entirely. The skill's structural guidance creates more surface area for violations, even though it explicitly forbids `SELECT *`. This highlights a design tension: richer architectural guidance may introduce failure modes that simpler instructions avoid.

Ceiling effects in Dockerfile. The Dockerfile domain shows no significant pseudocode advantage, likely because both formats achieve near-ceiling compliance (<4.1% failure rate). The rules themselves may be simple enough that any reminder suffices. The SQL, Chart, and Terraform domains, with higher failure rates and more nuanced rules (e.g., "COALESCE nullable dimensions to '(unknown)'", "muted color palette"), provide more room for format to matter.

Cross-family divergence. The pseudocode effect is statistically significant for Claude models in SQL ($\delta = 0.982$, large) and Chart ($\delta = 0.472$, medium), and trends positive in Terraform, but is not significant for GLM models individually (Table 8). Gemini 3.1 Pro shows a positive pseudocode direction in three of four domains, with the largest effects in Chart ($\delta = 0.247$) and Terraform ($\delta = 0.222$), though a notable exception is SQL where Gemini shows a large *negative* effect ($\delta = -0.704$, $p = 0.007$), meaning pseudocode significantly hurts compliance for this family–domain pair. A possible mechanism is over-literal interpretation of class-level structure: the SQL pseudocode skill defines rules within a `PerFileRules` class, and Gemini may apply per-file logic too aggressively—losing points on pipeline-level rules (e.g., `COALESCE`, deduplication) that are architecturally appropriate only in specific dbt layers rather than in every model file. The SQL evaluation metric, which averages pass rates across files, may amplify this effect. This aligns with He et al.'s finding that format preferences do not transfer across model families [5]. The three frontier models (Opus 4.6, GLM-5, Gemini 3.1 Pro) each show a different pattern: Opus strongly favors pseudocode, GLM-5 slightly favors Markdown, and Gemini favors pseudocode modestly overall but with a strong reversal in SQL. Format preferences are model-family-dependent, though the *direction* favors pseudocode in two of three families.

Reliability beyond the mean. RQ5 reveals that pseudocode's benefit extends beyond mean reduction to *variance reduction* for most models. The pooled variance ratio of 1.60 (Levene's $p = 0.002$) means Markdown runs have 60% more spread. Pseudocode narrows the HDI for four of five models, with the largest effect on Haiku 4.5 (HDI width 38.5% → 31.2%) and Opus 4.6 (17.3% → 10.7%). Practically, this manifests as a higher probability of meeting a production-quality threshold (<10% failure rate): Haiku 4.5 gains +22.2pp and Opus 4.6 gains +20.9pp under pseudocode. For teams deploying LLM agents at scale, reduced variance translates to fewer outlier failures and more predictable behavior.

Practical recommendations. For practitioners writing skill files:

(1) **Any skill file helps**. The 4.0× failure rate reduction from the baseline (RQ1) dwarfs the format effect.
(2) **Pseudocode offers a small additional advantage** in domains with complex, nuanced rules. For domains where skills already achieve near-perfect compliance (e.g., Dockerfile), format matters less.

(3) **The pseudocode advantage is model-dependent**. Some frontier models (Opus 4.6) benefit strongly, while others (GLM-5) show no preference or even slight reversal; Gemini shows a mixed pattern with a strong negative effect in SQL.
(4) **Semantic content matters more than format**. The gap between any-skill and no-skill is 9× larger than the gap between pseudocode and Markdown (25.0pp vs. 2.7pp).

Comparison to prior work. Mishra et al. [13] found 7–16 F1 point gains from pseudocode on NLP tasks using BLOOM and CodeGen. Our work extends this to structured-output generation using modern frontier models (2025–2026 vintage) and finds a smaller but consistent advantage. The smaller effect size is expected: our Markdown baseline is already a well-structured skill file (not plain text), making the comparison stricter. SkillsBench [7] found a 16.2pp skill presence effect; our 25.0pp pooled effect (33.4% → 8.4%) is comparable in magnitude, despite measuring compliance rather than pass rate.

6 THREATS TO VALIDITY

Internal validity. Automated evaluators may produce false positives or false negatives. We mitigated this by documenting edge cases in per-domain rubrics and iterating evaluator logic on pilot runs before the main experiment. Evaluators were finalized before data collection. The same author wrote both Markdown and pseudocode skills, which could introduce systematic bias; we verified that both formats encode the same rules through independent rule-by-rule comparison (Section 3.2), though the formats differ in structural richness (typed fields, enums, validation functions) beyond surface presentation.

Construct validity. Rule compliance is a proxy for output quality, not a direct measure. A Dockerfile that passes all 13 rules may still be functionally incorrect (e.g., wrong base image for the application). We include outcome checks (correct port, runtime match) for Dockerfile and Terraform, but these are secondary metrics.

External validity. We test four domains, all producing text-based structured output. Results may not generalize to interactive agent tasks, multi-turn conversations, or visual output. We test three model families; other families (GPT, Llama) may respond differently. The three tasks per domain provide limited coverage of each domain's full complexity space. Not all models are tested in all domains (Chart lacks GLM-4.7).

Reliability. The three original domains have 5 repetitions per cell for the four original models; Chart and Gemini 3.1 Pro use 3 repetitions. Fewer repetitions reduce statistical power for per-domain analyses. Some cells in the original domains have 4 or 6 repetitions due to calibration runs, which we include in the analysis. One Terraform run (Opus, Markdown, task 3, rep 5) was excluded because the CLI infrastructure repeatedly blocked the model's file-write attempts, producing an incomplete output unrelated to model capability; the remaining four repetitions of that cell scored consistently (12/13 rules). Gemini 3.1 Pro was added as a fifth model after the initial four-model experiment, using the same evaluation pipeline and identical skill files.

7 CONCLUSION

We presented the first controlled comparison of instruction format for LLM skill files. Across 737 runs spanning four structured-output domains and five models from three families, we find that (1) skill files provide a large and consistent compliance improvement over no-skill baselines, (2) pseudocode format offers a statistically significant additional advantage over Markdown prose ($p = 0.003$), though the effect size is small and varies by model family, and (3) pseudocode reduces output variability (variance ratio 1.60, $p = 0.002$), increasing the probability of meeting production-quality thresholds by up to 22 percentage points.

The practical implication is clear: writing skill files in pseudocode rather than Markdown is a low-cost intervention that reduces output defects by an additional 28% (9.8% → 7.1% failure rate) while also tightening the distribution for most models, with no downside beyond slightly longer input prompts. The format advantage is model-dependent: Claude Opus 4.6 benefits substantially (−44% relative reduction), Gemini 3.1 Pro achieves the lowest overall failure rate (4.1% with pseudocode), while GLM-5 shows no format preference. For practitioners authoring skill files for LLM agents, we recommend starting with any well-structured skill file (the largest effect), and considering pseudocode format for domains with complex constraints where both lower failure rates and predictable behavior matter.

Future work should extend this comparison to interactive agentic tasks with tool use, additional models (e.g., GPT, Llama), more domains (e.g., Kubernetes manifests, CI/CD pipelines), and user studies measuring skill file authoring effort. Ablation studies that isolate the contributions of type annotations, validation functions, and Python syntax would clarify whether the pseudocode advantage derives from structural priors or surface formatting. Hybrid approaches—such as Markdown with embedded JSON schemas or TypeScript interfaces—may achieve comparable compliance gains with lower authoring overhead.

REFERENCES

[1] Hyungjoo Chae et al. 2024. Language Models as Compilers: Simulating Pseudocode Execution Improves Algorithmic Reasoning in Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 22471–22502.
[2] Norman Cliff. 1993. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological Bulletin* 114, 3 (1993), 494–509.
[3] Bradley Efron and Robert J. Tibshirani. 1994. *An Introduction to the Bootstrap*. Chapman and Hall/CRC.
[4] Federico Errica, Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. 2025. What Did I Do Wrong? Quantifying LLMs' Sensitivity and Consistency to Prompt Engineering. In *Proceedings of the 2025 Conference of the North American Chapter*

of the Association for Computational Linguistics (NAACL). https://aclanthology.org/2025.naacl-long.73/
[5] Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X Wang, and Sadid Hasan. 2024. Does Prompt Formatting Have Any Impact on LLM Performance? *arXiv preprint arXiv:2411.10541* (2024).
[6] Prince Kumar, Rudra Murthy, Riyaz Bhat, and Danish Contractor. 2025. Training with Pseudo-Code for Instruction Following. *arXiv preprint arXiv:2505.18011* (2025).
[7] Xiangyi Li et al. 2026. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks. *arXiv preprint arXiv:2602.12670* (2026).
[8] Xinyu Liu et al. 2024. A Survey of Text-to-SQL in the Era of LLMs: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109* (2024).
[9] Yuanye Liu et al. 2025. Beyond Prompt Content: Enhancing LLM Performance via Content-Format Integrated Prompt Optimization. *arXiv preprint arXiv:2502.04295* (2025).
[10] Jun Lyu, He Zhang, Yusong Yuan, Lanxin Yang, Yue Li, and Manuel Rigger. 2026. Automatic Dockerfile Generation with Large Language Models. In *Proceedings of the 48th International Conference on Software Engineering (ICSE)*.
[11] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, et al. 2023. Self-Refine: Iterative Refinement with Self-Feedback. *Advances in Neural Information Processing Systems (NeurIPS)* (2023).
[12] Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language Models of Code are Few-Shot Commonsense Learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
[13] Mayank Mishra, Prince Kumar, Riyaz Bhat, Rudra Murthy, Danish Contractor, and Srikanth Tamilselvam. 2023. Prompting with Pseudo-Code Instructions. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 15178–15197. https://aclanthology.org/2023.emnlp-main.939/
[14] Lilian Ngweta, Kiran Kate, Jason Tsay, and Yara Rizk. 2025. Towards LLMs Robustness to Changes in Prompt Format Styles. In *Proceedings of the 2025 Conference of the North American Chapter of the Association for Computational Linguistics: Student Research Workshop*. https://aclanthology.org/2025.naacl-srw.51/
[15] Aske Plaat et al. 2025. Agentic Large Language Models, a Survey. *arXiv preprint arXiv:2503.23037* (2025).
[16] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2024. Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*.
[17] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2023. ProgPrompt: Program Generation for Situated Robot Task Planning using Large Language Models. *Autonomous Robots* 47 (2023), 999–1012.
[18] Brandon T Willard and Rémi Louf. 2023. Efficient Guided Generation for Large Language Models. *arXiv preprint arXiv:2307.09702* (2023).
[19] Renjun Xu and Yang Yan. 2026. Agent Skills for Large Language Models. *arXiv preprint arXiv:2602.12430* (2026).

A PROMPT TEMPLATES

All three conditions use the same task data (JSON object with task requirements). The conditions differ in the instruction prefix.

No-skill condition. The task JSON is followed by a domain-specific generic instruction (e.g., "Generate a Dockerfile. Output ONLY the Dockerfile.").

Markdown condition. The skill file (Markdown prose with headings, tables, and checklists) is prepended to the task JSON.

Pseudocode condition. The skill file (Python pseudocode with dataclasses, enums, and validation functions) is prepended to the task JSON.

Full prompt templates and skill files are available in the repository under `papers/1-pseudocode-format/domains/{domain}/skill` and `papers/1-pseudocode-format/prompts/`.

B PER-DOMAIN RULE DETAILS

Chart / Vega-Lite (15 scored rules). (1) Muted/professional color palette, (2) Accent colors ($\leq$2 highlights), (3) Accessibility (no red+green conflict), (4) Single chart type consistency, (5) Insight-driven title ($\geq$25 chars), (6) Source citation, (7) Sans-serif font, (8) Data labels when $\leq$8 points, (9) Y-axis origin at zero, (10) Minimal spines, (11) Subtle grid colors, (12) Units in $\geq$2 locations, (13) Key annotations, (14) Direct labels vs. legend, (15) Chart dimensions.

Dockerfile (13 scored rules). (1) Specific image tag on every FROM, (2) Non-root USER, (3) No secrets in ENV/ARG, (4) Multi-stage build, (5) WORKDIR before COPY/RUN, (6) Dependencies before source code, (7) Combined RUN instructions ($\leq$2 adjacent), (8) apt-get best practices, (9) HEALTHCHECK present, (10) EXPOSE documented, (11) LABEL metadata, (12) Exec form CMD/ENTRYPOINT, (13) No unnecessary ADD.

SQL / dbt (12 scored rules). (1) Keywords uppercase, (2) One clause per line, (3) Table aliases, (4) Column aliases with AS, (5) No SELECT *, (6) Comment header, (7) LEFT JOIN only, (8) COALESCE to '(unknown)', (9) ROW_NUMBER dedup, (10) One CTE per file, (11) Jinja ref(), (12) Layer naming conventions.

Terraform (13 scored rules). (1) snake_case naming, (2) Variable descriptions, (3) Variable type constraints, (4) Outputs defined, (5) Tags on taggable resources, (6) Variable grouping, (7) File structure hints, (8) No hardcoded IDs, (9) Provider version pinned, (10) Backend configured, (11) Sensitive values marked, (12) Data sources for lookups, (13) Locals block present.

C PER-RULE PASS RATE BREAKDOWN

Tables 11–14 show pass rates for every scored rule by condition, revealing where the pseudocode advantage concentrates and where skills can *hurt* compliance.

Largest pseudocode advantages. SQL rules 2 and 5 (one clause per line, no `SELECT *`) show the largest pseudocode–Markdown gaps (+53.3pp and +52.9pp). Both rules fail together because Markdown-guided models create staging models with `SELECT * FROM source` on a single line—a shortcut that violates both rules simultaneously. Pseudocode models avoid this pattern (98.6% and 98.2% pass). In the Chart domain, rules 8 (data labels) and 12 (unit labels) show +33.3pp and +41.2pp gaps; the pseudocode validation functions explicitly check these. Terraform rules 2–4 (variable descriptions, types, outputs) show +13–15pp, consistent with pseudocode's

typed dataclass fields making these requirements harder to miss.

Largest Markdown advantages. Chart rule 2 (accent color limit) shows the largest Markdown advantage (−36.6pp). The Markdown skill uses a clear table for this rule, while the pseudocode encodes it as a validation function that may be less salient. SQL rule 9 (ROW_NUMBER dedup, −12.5pp) and Terraform rule 1 (naming, −10.0pp) also favor Markdown modestly.

Skill-induced complexity in SQL.. Rules 2 and 5 illustrate a subtle effect: the skill instructs a multi-layer dbt pipeline with staging models, creating more opportunities for `SELECT *` violations than monolithic queries. Despite the skill explicitly forbidding `SELECT *`, most models (except GLM-5) take a shortcut in staging CTEs. This correlation is near-perfect ($r = 0.996$ between rule 2 and rule 5 scores).

Table 11: Chart: per-rule pass rate (%) by condition.

#	Rule	None	MD	PC	PC−MD
1	Muted palette	5.9	100.0	100.0	0.0
2	Accent ≤2	100.0	95.8	59.3	−36.6
3	Color accessibility	94.1	100.0	100.0	0.0
4	Single chart type	100.0	100.0	100.0	0.0
5	Insight title	77.8	100.0	100.0	0.0
6	Source citation	100.0	100.0	100.0	0.0
7	Sans-serif font	100.0	100.0	100.0	0.0
8	Data labels	3.7	66.7	100.0	+33.3
9	Y-axis origin zero	100.0	95.5	100.0	+4.5
10	Minimal spines	—	100.0	100.0	0.0
11	Subtle grid	—	100.0	100.0	0.0
12	Units in ≥2 loc	100.0	58.8	100.0	+41.2
13	Key annotations	100.0	88.9	100.0	+11.1
14	Direct labels	85.2	95.5	100.0	+4.5
15	Chart dimensions	—	100.0	100.0	0.0

Table 12: Dockerfile: per-rule pass rate (%) by condition.

#	Rule	None	MD	PC	PC−MD
1	Specific tag	92.1	100.0	98.4	−1.6
2	Non-root USER	66.7	100.0	100.0	0.0
3	No secrets	98.4	100.0	100.0	0.0
4	Multi-stage	88.9	100.0	100.0	0.0
5	WORKDIR first	82.5	96.8	93.7	−3.2
6	Deps before source	60.3	57.1	63.5	+6.3
7	Combined RUN	88.9	92.1	90.5	−1.6
8	apt-get practices	98.4	100.0	100.0	0.0
9	HEALTHCHECK	69.8	100.0	100.0	0.0
10	EXPOSE	96.8	100.0	100.0	0.0
11	LABEL metadata	19.0	100.0	100.0	0.0
12	Exec form CMD	96.8	100.0	100.0	0.0
13	No ADD	98.4	100.0	100.0	0.0

Table 13: SQL (dbt): per-rule pass rate (%) by condition. Rules 2 and 5 are correlated because `SELECT * FROM source` violates both.

#	Rule	None	MD	PC	PC−MD
1	Keywords uppercase	90.8	100.0	100.0	0.0
2	Clause per line	94.6	45.3	98.6	+53.3
3	Table aliases	67.6	99.5	99.4	−0.2
4	Column aliases (AS)	87.4	97.9	97.9	0.0
5	No SELECT *	95.4	45.3	98.2	+52.9
6	Comment header	24.6	98.8	90.5	−8.3
7	LEFT JOIN only	36.7	97.8	98.3	+0.6
8	COALESCE unknown	0.0	44.4	40.3	−4.0
9	ROW_NUMBER dedup	31.7	72.1	59.6	−12.5
10	One CTE per file	70.3	99.1	99.5	+0.5
11	Jinja ref()	0.0	98.3	95.0	−3.3
12	Layer naming	0.0	98.3	96.7	−1.7

Table 14: Terraform: per-rule pass rate (%) by condition.

#	Rule	None	MD	PC	PC−MD
1	snake_case naming	33.3	94.9	83.3	−11.6
2	Variable descriptions	61.7	76.3	90.0	+13.7
3	Variable types	76.7	76.3	90.0	+13.7
4	Outputs defined	56.7	78.0	91.7	+13.7
5	Tags on resources	68.3	100.0	100.0	0.0
6	Lifecycle ignore	95.0	100.0	100.0	0.0
7	Variable separation	95.0	98.3	100.0	+1.7
8	File structure	95.0	100.0	100.0	0.0
9	No hardcoded IDs	43.3	30.5	23.3	−7.2
10	Provider pinned	70.0	100.0	100.0	0.0
11	Backend configured	0.0	88.1	100.0	+11.9
12	Sensitive marked	88.3	91.5	95.0	+3.5
13	Data sources	90.0	98.3	91.7	−6.6
14	Locals block	15.0	89.8	98.3	+8.5

D REPRODUCTION

All materials are available at https://github.com/aictrl-dev/skill-md-research:

- `papers/1-pseudocode-format/domains/{domain}/skills/` — Matched skill file pairs
- `papers/1-pseudocode-format/domains/{domain}/test-data/` — Task JSON files
- `papers/1-pseudocode-format/domains/{domain}/evaluate_*.p` — Automated evaluators
- `papers/1-pseudocode-format/domains/{domain}/results/scor` — Raw scored data
- `papers/1-pseudocode-format/paper/compute_stats.py` — Cross-domain statistics
- `papers/1-pseudocode-format/scripts/generate_figures.py` — Figure generation
- `papers/1-pseudocode-format/scripts/variability_analysis.` — Reusable variability analysis (HDI, Levene, threshold rates, violin plots)