



One-to-One: Evaluating AI Personalization in Marketing Systems

Vikram Sivashankar

Conversion Labs · September 28, 2026

Abstract

An agent that personalizes a marketing campaign must generate a program whose output remains correct across many recipients. We study this problem in a marketing platform where users describe the desired behavior and an agent writes a Liquid template using the workspace schema and validation tools. We introduce One-to-One, an evaluation that executes generated templates on synthetic contact records and checks the resulting messages and structured outputs. An audit of a 150-task development suite identifies both valid answers rejected by implementation-specific checks and incomplete answers accepted by permissive checks. We then freeze the generator and evaluate three model routes on 36 new requests, with reference programs verified on all 24 contacts and a stricter output grader. Across 324 completed attempts, final task success is 97.2% for Claude Opus 5, 95.4% for Kimi K3, and 87.0% for GLM 5.3 Flash. Within-run revisions turn 17 incorrect first drafts into correct final answers, but 22 locally accepted templates still fail the output evaluation. The study separates inference failures from completed incorrect answers, records the first draft and final answer, and reports observed latency and gateway-reported cost. The results characterize personalization program generation in one application environment, rather than recipient preferences or campaign effectiveness.

1 Introduction

A marketer might ask an agent to mention the customer’s renewal date, use the right account team for their region, and omit any details that are missing. The resulting message must work for a contact with a complete record, another without a company, and another whose company has several opportunities. Writing one plausible example is insufficient: the agent must express the requested behavior as a program that will run for each recipient.

This is a practical form of language-to-code generation. The target language is small, but the operating environment matters. Fields differ between workspaces. Related records can be absent or contain multiple entries. JSON must preserve types and escaping, links must encode values, and edits must preserve text the user did not ask to change. A valid template can violate any of these requirements without raising an execution error.

We study an existing Liquid generator integrated into a marketing platform. The agent receives a workspace schema, can inspect fields and values, and submits a template to a checker that returns errors for revision. The deployed renderer and the local checker are different implementations. This creates two evaluation questions: does the agent produce a template the application accepts, and does the accepted template produce what the user requested?

One-to-One evaluates the second question through executed outputs. We first audit a development comparison across eight models, then evaluate a frozen generator on new requests with complete expected outputs for the synthetic contact panel. We record each submitted draft, making it possible to compare the first submission with the final answer in the same run. Our contributions are the application-specific evaluation and its inspectable fixtures, an empirical audit of scoring errors, and a controlled description of model behavior, revision, latency, and cost under a shared generator.

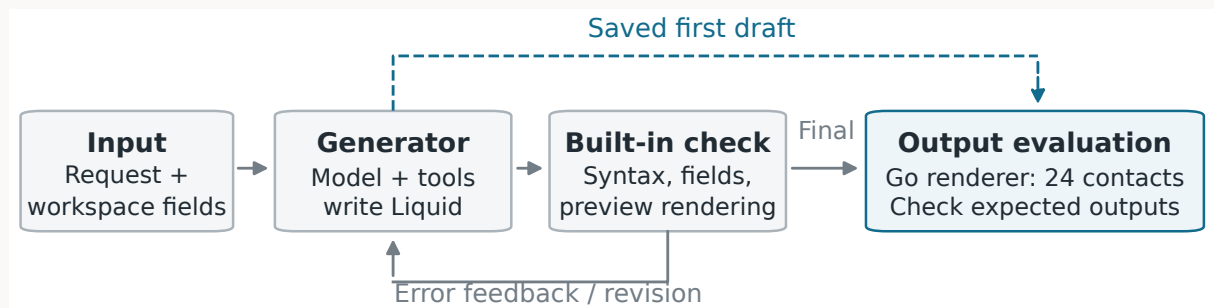


Figure 1: Evaluation process. The first submitted draft and final answer are saved and evaluated against the same expected outputs for 24 synthetic contacts. Built-in acceptance and output correctness are separate checks.

2 Application and generator

The platform exposes Liquid in editors for messages, subjects, HTML, JSON, URLs, and typed field values. A user can request a new template or describe a change to an existing one. The generator receives the request, destination surface, current template when applicable, and the workspace’s contact, company, opportunity, and other available fields.

The generator retrieves a field catalog and campaign tokens, constructs a schema description, and supplies a reference for the platform’s Liquid dialect. Its tools support field lookup, inspection of distinct values, record search, form inspection, and template submission. The submission tool parses the tem-

plate, checks field paths and supported constructs, and renders two local examples with populated and missing values. A rejected submission returns its error to the model. Templates emitted as plain text pass through the same checker before acceptance.

The checker uses liquidjs configured to approximate the application's Go renderer. The evaluation renders saved templates through the Go implementation in a staging workspace. These executions do not send messages to customers. Acceptance by the local checker and correctness in the evaluation are therefore separate observations (Figure 1).

We retain the generator's six-step limit, three-rejection limit, 4,096 output-token cap, and requested minimal reasoning setting. Provider-specific settings can still differ. For the new evaluation, we increase the wall-clock deadline from 30 to 120 seconds and add passive logging of submissions and response metadata. No prompt, schema-presentation, or tool changes are made after the generator freeze.

3 Evaluation design

3.1 Development suite and scorer audit

The development suite contains 150 synthetic requests spanning basic substitutions, conditional copy, collections, dates, JSON, URLs, edits, workflow events, and unsupported requests. Two tasks lack required forms, leaving 148 executable tasks. Eight models receive three attempts per task. The tasks were used during development; they are not an independent test set. The preserved records contain 3,552 original attempts and separately dated replacements for unavailable or incomplete inference.

The original grader combines output equality, substring and regular-expression checks, JSON predicates, and requirements for particular template paths or constructs. We replay all 3,566 scored records among the original and replacement attempts: every recorded verdict reproduces. We then inspect task requirements, probe the grader with altered outputs, and review a fixed sample of 32 answers, balanced across models and original passing/failing status. This review is AI-assisted, not independent human annotation or a population estimate of grader accuracy.

The audit finds three actionable problems. First, six correct opportunity lists are rejected because the template uses a working field alias rather than the path named by the grader. Second, eight answers pass a country-only branch despite adding unrequested text. Third, three workflow tasks require the identity of a changed field, but their preview fixture provides old and new values without a populated field identity. Their checks can reward incomplete answers and penalize justified refusals. Removing all three affected tasks and applying the two output-grounded corrections changes the ordering of several middle-ranked models. We retain both original and sensitivity results rather than presenting the corrected development scores as independent evidence.

Figure 2 shows the development scores before and after these changes. Opus and Kimi remain the highest-scoring models. GPT-5.6 moves from 92.1% to 94.5%, illustrating the effect of correcting the grader rather than changing the model.

Other checks leave requirements untested. Whitespace normalization removes line structure, some dates are checked only by shape, and refusal status can reflect a generic harness fallback. These findings motivate explicit output contracts and reference validation for the new evaluation.

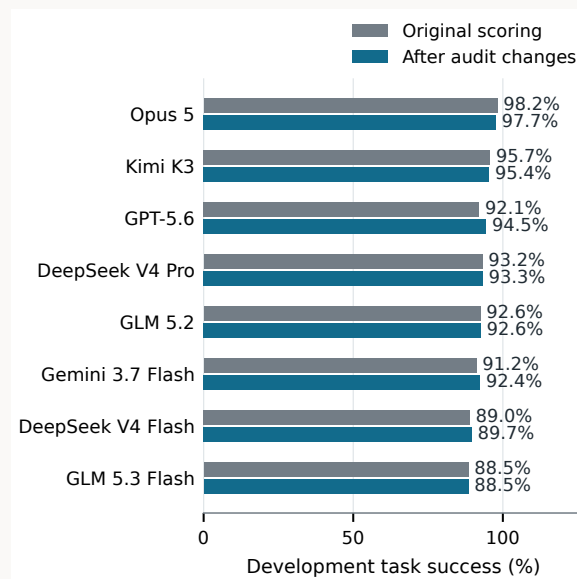


Figure 2: Eight-model development comparison under the original scoring and targeted audit changes. Original scores cover 148 executable tasks with 442-444 eligible attempts per model. The audit removes three invalid-fixture tasks, leaving 433-435 attempts, and corrects the two checks described in the text. This is post-hoc sensitivity, not a full semantic regrade or held-out evaluation.

3.2 New requests and reference validation

After freezing the generator, we author 36 new requests in six groups of six: routing, text composition, structured output, opportunity collections, fixed dates and numbers, and URLs or edits. They test new combinations of supported behavior, not a new workspace distribution. The authoring process can use lessons from the development audit, but the new tasks and generator remain unchanged after the evaluation freeze. One output-scoring defect is corrected after collection, as described below.

The existing synthetic panel contains 24 contacts. It includes missing names and companies, different lifecycle stages and regions, mixed capitalization, accented characters, and companies with zero or multiple opportunities. We snapshot the fields used by the tasks and compute an expected output for every contact. Unlike the development grader's selected-anchor checks, the new grader checks all 24 outputs per request.

Each task has a reference Liquid program and a separately implemented expected-output function, without consulting candidate-model answers. All 864 reference outputs pass both production rendering and the task grader; each reference program also passes the frozen local checker. We verify that 2,557 deliberately incorrect output mutations are rejected. Mutations cover applicable missing or extra keys, wrong types or values, reordered arrays and URL parameters, extra content, and collapsed lines. These tests establish the checked properties, not complete semantic correctness for all possible contacts.

Workflow requests are excluded from this new set because the available workflow fixture was not validated. Tasks use fixed date cutoffs rather than a moving current date. Expected behavior for missing values, punctuation, ordering, and types is explicit. Task definitions, fixture snapshot, reference outputs, scorer, and generator hashes are retained before generation.

3.3 Scoring and paired drafts

A request passes only if its output is correct for every contact and production rendering succeeds. Text comparison preserves internal whitespace and line breaks, normalizing only CRLF and outer whites-

pace. JSON comparison requires the full object structure, exact values and types, and ordered arrays while ignoring object key order. URL comparison checks scheme, host, path, ordered decoded query parameters, and fragment, allowing equivalent percent encoding. It also enforces the requests' encoding requirements and rejects raw whitespace, malformed escapes, unencoded reserved characters in values, and trailing empty delimiters. The grader does not require a particular Liquid implementation.

During final review, we find that the frozen URL grader accepts a raw space because it compares decoded parameters. We correct this defect and rescore every saved first and final answer, preserving the original scores and documenting the amendment. Exactly one GLM first-draft verdict changes from pass to fail; no final verdict changes. There are no task changes or model reruns. An independent Python implementation agrees with the corrected TypeScript scorer on all 15,552 first/final contact checks.

We save the first submitted draft before the checker evaluates it, subsequent submissions, and the final answer. Both the first draft and final template are rendered and scored with the same oracle. This is a paired measurement of revisions within the existing agent. It is not a separately prompted no-tool baseline: the first draft can already use lookup tools and the agent knows it will be checked. A completed run without a template fails the corresponding output metric.

3.4 Models, missing answers, and resources

We select Claude Opus 5 and Kimi K3, the two highest-scoring routes in the development comparison, and GLM 5.3 Flash, its fastest route. We schedule three repetitions of every request for each model, yielding 324 model-task attempts, in a fixed randomized order. These routes provide an informative quality and latency contrast; they are not an exhaustive ranking of current models.

Every attempt is logged. A provider or transport failure is not scored as an incorrect answer. Interrupted attempts remain visible and can receive bounded retries, retaining the first completed outcome. An incorrect completed template is never rerun to obtain a passing answer. Explicit refusals and exhausted output, step, or checker budgets are retained as unsuccessful outcomes when the task is feasible. Missing answers and completed coverage are reported separately.

Builder latency includes schema preparation, model calls, and internal revisions. It excludes the later benchmark rendering and retry delays. First-draft latency ends when the first template is submitted. We retain usage, serving-provider metadata, response model identifiers, and gateway-reported charges, including the available charges for retry attempts. These charges are distinct from a separate conservative spending guard and from the historical sweep, whose billed costs are unknown.

We resample the 36 tasks 10,000 times, keeping all three repetitions together, for descriptive 95% intervals. We also assess paired revision gains by resampling the six request groups. Repetitions are not treated as independent tasks. Intervals describe variation within this constructed set, not uncertainty over real customer requests.

4 Results

4.1 Output correctness

All 324 scheduled attempts complete, with no connectivity exclusions or retries. Every final template is accepted by the local checker. Table 1 reports strict task success: one incorrect contact output is enough to fail an attempt.

Model	First draft	Final	Final 95% interval
Opus 5	102/108	105/108 (97.2%)	92.6–100%
Kimi K3	95/108	103/108 (95.4%)	90.7–99.1%
GLM Flash	88/108	94/108 (87.0%)	77.8–94.4%

Table 1: Task success on the new requests. A pass requires correct output on all 24 contacts. Each model has 108 completed attempts.

Opus has the highest observed task success, but its interval overlaps Kimi’s substantially. These data do not establish a stable ordering between the two. The gap between Opus and GLM is 10.2 percentage points on this set. At the contact level, Opus produces 2,528 correct outputs out of 2,592, Kimi 2,549, and GLM 2,405. Kimi’s higher contact-level total alongside its lower task-level score illustrates why the aggregation matters: one error repeated across many contacts differs from several templates with narrower failures.

4.2 Revision and remaining failures

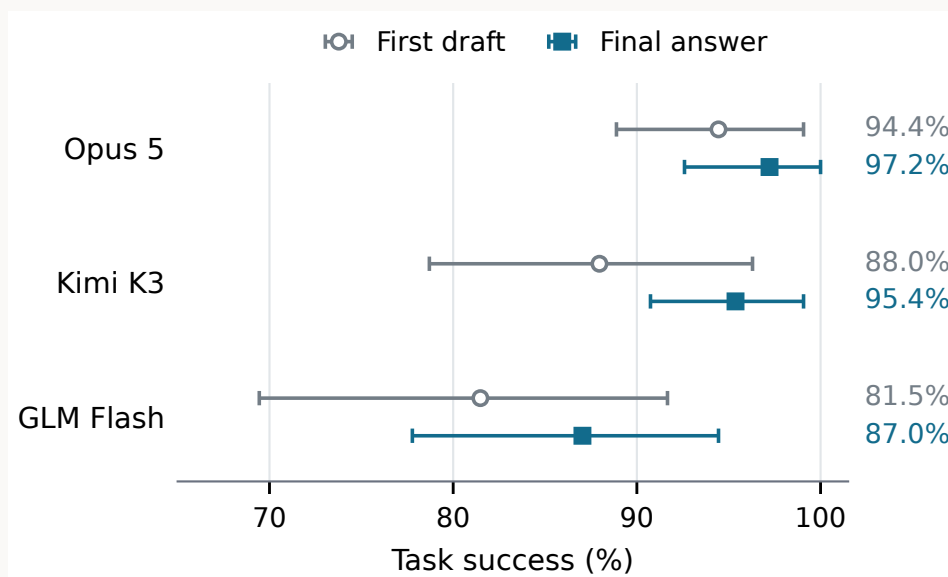


Figure 3: First-draft and final task success. Whiskers show 95% task-cluster bootstrap intervals over 36 requests, keeping three repetitions together. These are paired observations within the same runs.

Figure 3 compares first-draft and final task success. First drafts pass 285 of 324 attempts (88.0%); final answers pass 302 (93.2%). Three Opus, eight Kimi, and six GLM attempts move from incorrect to correct. No correct first draft becomes an incorrect final answer under the corrected grader. The generator records 30 checker rejections across 26 attempts with multiple submissions. The improvement is associated with the existing revision loop, not an isolated causal estimate of checker value.

Task-cluster intervals for the revision gain are 0.0–7.4 percentage points for Opus, 1.9–13.9 for Kimi, and 0.9–12.0 for GLM. Resampling the six request groups instead gives 0.0–8.3, 0.0–16.7, and 1.9–9.3, respectively. With only six groups, these intervals are descriptive and sensitive to the constructed task mix.

The 22 accepted but incorrect final answers comprise ten structured-output failures, five URL/edit failures, three routing failures, two collection failures, and two composition failures. None occurs in the

fixed-date and number group. Twenty-one render successfully but violate the requested output; one fails in the Go renderer despite local acceptance.

Three examples illustrate the distinction. Two Opus answers encode the equals sign in `tier=growth`, producing `tier%3Dgrowth` as a parameter name instead of the requested key/value pair. A Kimi answer produces one array element containing both tags and escaped quotes, rather than two separate tag values. Two GLM routing answers read the lead-status field where the request requires lifecycle stage, so qualified contacts receive the wrong instruction. All are accepted locally.

The production-only failure occurs when a GLM template sorts `company.opportunities` by name. The Go renderer rejects the record query, although the local checker accepts it. We reproduce the Liquid syntax error separately; it is not a provider or transport failure. All 22 failed attempts are also detectable on the six original anchor contacts. The larger contact panel provides more complete output evidence here, but does not identify additional failed tasks beyond those anchors.

4.3 Latency and observed cost

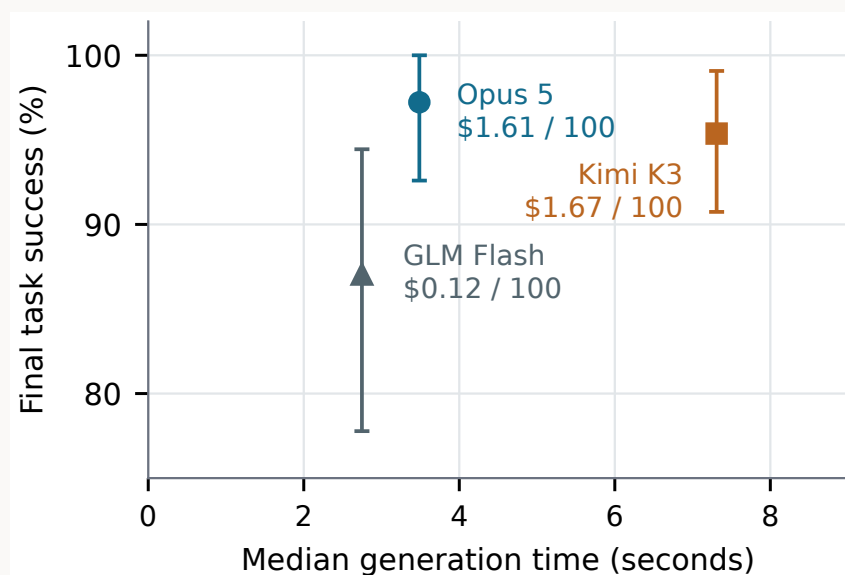


Figure 4: Observed speed, quality, and model cost. Whiskers show 95% task-cluster intervals for final success. Labels give gateway-reported dollars per 100 attempts. Times and costs reflect the cached, repeated-workspace workload.

Model	Median (s)	p95 (s)	Cost / 100
Opus 5	3.49	12.55	\$1.61
Kimi K3	7.31	17.18	\$1.67
GLM Flash	2.75	9.98	\$0.12

Table 2: Builder latency and gateway-reported cost for the new evaluation. Rendering is measured separately from generation.

Figure 4 shows the observed speed and quality tradeoff alongside model cost. The complete evaluation incurs \$3.67 in gateway-reported model charges. GLM is fastest and cheapest in this run, with the lowest task success. Opus and Kimi have similar observed cost despite different latency. Median first-draft times are 3.46, 7.18, and 2.37 seconds, respectively.

These are warm, repeated-workspace measurements. Reported cache reads account for 96.9% of Opus input tokens, 63.9% of Kimi input tokens, and 47.7% of GLM input tokens. The costs therefore should not be read as uncached pricing, and the latency does not predict a cold first request. Gateway charges are recorded usage costs, not an independently reconciled invoice. A post-run snapshot confirms that the 24 fixture records are unchanged.

5 Implications for a marketing agent

The practical requirement is that the same generated program behaves correctly for different recipients. Testing only a populated preview gives weak evidence for missing-data branches. Checking that JSON parses says nothing about whether a number became a string. Inspecting a template path can reject a correct alias. These are different failure modes and require different evidence.

The observed revisions support retaining a feedback loop: 17 runs finish correctly after an incorrect first submission. The remaining 22 failures show the limit of the current checker. It validates syntax, fields, and a small preview, but does not fully establish that the requested behavior is implemented. For this system, the next concrete checker improvements are URL construction, JSON values and types, routing field selection, and consistency with production collection operations.

Choosing a model also depends on the unit of failure the product cares about. A marketer deploys a template across recipients, so our primary score requires all tested outputs to be correct. Contact-level accuracy is still useful for describing the reach of an error, but can hide a template that fails only for contacts with missing fields. The two measures give different orderings for Opus and Kimi in this experiment.

The evaluation also separates output quality from availability. Treating an unavailable inference endpoint as an incorrect template would confound provider reliability with program generation. Conversely, retrying completed wrong answers until one passes would exaggerate quality. Keeping immutable attempt records makes both choices inspectable.

The reference programs and explicit output contracts are useful engineering artifacts beyond the model comparison. They make renderer compatibility and scoring changes testable without another model run. When a product changes its schema, renderer, or checker, the same inputs and saved templates can distinguish changes in application behavior from changes in generation.

6 Related work

HumanEval evaluates generated programs with execution tests ([Chen et al., 2021](#)). One-to-One applies functional evaluation to a constrained language embedded in a marketing application, where output formatting and workspace-specific records are part of the contract. Spider studies schema-grounded text-to-SQL across databases ([Yu et al., 2018](#)); our study shares schema grounding but does not establish cross-workspace generalization.

EvalPlus shows that stronger tests can change conclusions about generated-code correctness ([Liu et al., 2023](#)). Our scorer audit demonstrates a related issue in an application benchmark: both permissive output checks and implementation-specific restrictions can distort measured quality. Work on self-debugging studies how execution and feedback support program revision ([Chen et al., 2024](#)). Our draft comparison measures revisions in the existing product loop rather than introducing a new self-debugging method.

LaMP evaluates personalized classification and generation using user profiles (Salemi et al., 2024). Our use of personalization concerns executable customization from structured customer records. We measure whether the program follows the requested transformation, not whether the resulting prose matches a recipient’s preferences or persuades them.

7 Conclusion

One-to-One measures whether a personalization agent writes a program that follows a marketing request across synthetic customer records. On 36 new requests, three frozen generator configurations achieve 87.0–97.2% task success. Revisions recover 17 initially incorrect drafts, while 22 accepted templates remain incorrect. These findings identify a concrete gap between application acceptance and requested behavior. Executing templates against explicit output contracts makes that gap measurable and supplies regression cases for improving the agent and its checker.

Limitations

The new evaluation contains 36 synthetic requests on one existing synthetic workspace. It is held out from generator development, but not from knowledge of the application’s task families or development failures. The fixtures do not cover every boundary value or every customer schema. Three models and three repetitions provide a bounded comparison, not a general leaderboard.

Task authoring, reference construction, and qualitative review are AI-assisted. Agreement between an executable oracle and a reference program does not replace independent expert review. Mutation tests cover specified properties; they cannot establish that every natural-language interpretation has been captured.

The within-run comparison measures the difference between submitted drafts and final outputs. It does not isolate the value of the prompt, schema, lookup tools, checker, or additional inference separately. Provider routes, cache state, and serving revisions can affect cost and latency. The generator source is frozen; the staging service remains an external dependency.

The URL scoring amendment was made after observing model outputs; original and corrected scores are preserved. The historical study has incomplete provenance and known scoring weaknesses. Its scores and post-hoc corrections are reported as development evidence, separately from the frozen evaluation. Neither study measures customer adoption, recipient satisfaction, deliverability, or conversion lift.

Ethics and reproducibility

The contact panel is synthetic and uses a reserved test email domain. Evaluation renders templates without sending messages. Personalization systems can misuse sensitive attributes or generate misleading claims; this study does not evaluate those behaviors and its task pass rates should not be used as a safety certification.

The supplementary package contains task requests, reference programs, synthetic fixture values, saved outputs, scoring code, analysis, protocol, source hashes, and cost records. Scoring and analysis run on saved evidence with Python’s standard library. Live generation and rendering depend on private application services. The materials have not yet been publicly archived.

References

- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. [Evaluating large language models trained on code](#). *arXiv preprint arXiv:2107.03374*.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. [Teaching large language models to self-debug](#). In *The Twelfth International Conference on Learning Representations*.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. [Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation](#). In *Advances in Neural Information Processing Systems*, volume 36.
- Alireza Salemi, Sheshera Mysore, Michael Bendersky, and Hamed Zamani. 2024. [LaMP: When large language models meet personalization](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7370–7392. Association for Computational Linguistics.
- Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. [Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task](#). In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921. Association for Computational Linguistics.