From Plotting Primitives to Semantic Specifications: Altair as a Token-Efficient Interface for Agentic AI

Eduardo Di Santi
College of Engineering and Applied Science
eduardo.disanti@colorado.edu

Abstract

Agentic AI systems increasingly perform data analysis workflows that require generating executable visualizations. In primitive-based plotting libraries, agents must infer the semantic intent of a visualization and also synthesize the procedural code needed to construct figures, axes, legends, layouts, and interactions. This paper studies whether a declarative visualization grammar reduces this interface-induced procedural burden. We compare Matplotlib, as a primitive-based plotting interface, with Altair/Vega-Lite, as a declarative visualization interface. Across 11 visualization tasks, two library conditions, three repeated runs, and four successfully evaluated OpenAI models, we measure first-pass generation tokens, generated code lines, execution success, repair-token behavior, and repeat-run variability. The results show no consistent Altair advantage for low-complexity charts. However, for medium and higher-complexity tasks, Altair reduces first-pass generation tokens in most conditions, with aggregate reductions of 22.6% for medium tasks, 12.1% for medium-high tasks, and 31.1% for high-complexity tasks. Code-line reductions follow the same pattern. These results suggest that token efficiency is not merely a syntactic property of a library. It emerges from the alignment between task semantics, tool semantics, and model capability.

1 Introduction

Agentic AI systems are increasingly used not only to answer questions, but to perform analytical workflows: inspecting data, generating code, calling tools, debugging failures, modifying outputs, and communicating results [1]. This view is consistent with recent work on language models as reasoning-and-acting systems, where models interleave reasoning, tool use, code generation, and environment feedback [11, 5]. Visualization is a central component of these workflows.

Primitive-based visualization systems require the agent to operate at a lower level of abstraction than the analytical task itself. This distinction is visible when comparing imperative plotting environments such as Matplotlib [2] with grammar-based or declarative approaches such as ggplot2, Vega-Lite, and Altair [8, 3, 6]. When primitive-based plotting libraries are used, the agent must solve both the semantic problem of what should be visualized and the procedural problem of how to construct the chart.

This paper starts from a simple observation: before generating a visualization, an agent must already solve the semantic problem. It must determine what variables matter, how they relate, whether they are quantitative, ordinal, nominal, or temporal, whether aggregation is needed, and which visual encoding is appropriate. This reasoning is unavoidable. However, the procedural problem of instantiating figures, axes, marks, legends, scales, layouts, labels, and interaction handlers is largely imposed by the plotting interface.

The central claim of this paper is therefore:

> *Altair reduces token consumption in agentic AI not merely because its code can be shorter, but because it provides semantic visualization primitives that can reduce unnecessary procedural reasoning in the agentic loop.*

2 Motivation

Consider a request such as:

> Show the monthly revenue trend, colored by product category, with a tooltip and an interactive filter.

The agent must first solve the semantic task: identify `month` as temporal, `revenue` as quantitative, `product category` as nominal, choose a time-series representation, map color to category, and represent tooltip and filtering behavior as interaction semantics.

Once this has been solved, a primitive-based backend still requires additional procedural reasoning. The agent must decide how to create figures, configure axes, loop over groups, assign colors, construct legends, attach interaction handlers, and manage layout. In contrast, Altair allows much of the already-resolved semantic structure to be mapped directly into a declarative grammar. Thus, the relevant issue is not simply code length. The deeper issue is the size and structure of the agentic action space.

3 Semantic Reasoning vs. Procedural Plotting

We decompose visualization generation into three components:

$$C_{\text{viz}} = C_{\text{sem}} + C_{\text{proc}} + C_{\text{repair}}, \tag{1}$$

where C_{sem} is the cost of semantic reasoning, C_{proc} is the cost of procedural chart construction, and C_{repair} is the cost of debugging, correction, and iterative repair.

The semantic cost is unavoidable. Any agent must determine the analytical intent of the visualization. However, the procedural cost depends strongly on the visualization interface. Primitive-based libraries increase C_{proc} because the agent must synthesize plotting programs. Declarative libraries such as Altair reduce C_{proc} by exposing marks, encodings, transformations, and interactions as first-class semantic primitives.

For the empirical part of this report, we distinguish between first-pass generation cost and repair cost:

$$T_{\text{agentic}} = T_{\text{initial}} + T_{\text{repair}}. \tag{2}$$

The primary metric is T_{initial}, the observed token cost of the initial generation. Because internal reasoning tokens are not consistently observable across models and APIs, we use first-pass observable generation tokens as the primary metric, following the broader distinction between reasoning processes and observable model outputs in language-model workflows [7, 11]. Repair tokens are reported separately because they depend not only on the visualization interface, but also on package versions, API compatibility, local execution behavior, and repair-loop implementation.

4 Altair as an Agentic Visualization Interface

Altair is a Python interface to Vega-Lite, which builds on the Vega visualization grammar and provides a declarative grammar of interactive graphics [4, 3, 6]. Instead of specifying how to render a chart step by step, the user specifies what the chart represents. This places Altair in the tradition of grammar-based visualization systems, where the central abstraction is the mapping from data to visual encodings rather than a sequence of drawing commands [9, 10, 8].

A typical Altair chart has the form:

```
alt.Chart(data).mark_line().encode(
    x='date:T',
    y='value:Q',
    color='category:N'
)
```

This structure aligns closely with the semantic reasoning already performed by an agent. The agent has identified fields, types, and encodings. Altair then provides a compact representation in which these decisions can be expressed directly.

5 Hypotheses

We evaluate the following hypotheses:

H1: Complexity-dependent token reduction. Altair reduces first-pass generation tokens relative to Matplotlib primarily for medium and high-complexity visualization tasks, rather than uniformly across all tasks.

H2: Procedural-burden reduction. Altair reduces generated code length for non-trivial visualizations because the agent can express semantic encodings directly instead of constructing procedural plotting logic.

H3: Model-dependent declarative advantage. The benefit of a declarative visualization interface is model-dependent. More capable models may exploit the semantic structure of the grammar more consistently.

H4: Repair behavior as a separate reliability dimension. Repair tokens should be reported separately from first-pass generation tokens because they capture execution robustness, version compatibility, and repair-loop behavior in addition to representational burden.

6 Methodology

The objective of the empirical evaluation is to measure whether a declarative visualization interface reduces the token cost of agentic visualization generation compared with a primitive-based plotting interface. The experiment isolates the visualization library as the primary independent variable while keeping the dataset schema, task wording, execution protocol, and evaluation metrics fixed within each model condition.

6.1 Experimental Conditions

We compare two visualization interfaces: Matplotlib as the primitive-based plotting condition and Altair/Vega-Lite as the declarative visualization condition. Each visualization task is executed under both conditions.

Table 1: Experimental conditions.

Condition	Visualization interface	Description
A	Matplotlib	Primitive-based plotting interface
B	Altair / Vega-Lite	Declarative visualization specification

6.2 Models, Runs, and Exclusions

The benchmark was executed with OpenAI language-model APIs. Four models produced valid experimental outputs: `gpt-4o-mini`, `gpt-4.1-mini`, `gpt-5.4-mini`, and `gpt-5.5`. A code-specialized

model, `gpt-5.3-codex`, was also attempted, but all 66 calls failed at the API/harness level and produced no valid generations. It is therefore excluded from the empirical tables.

The valid benchmark contains 264 generated visualizations:

$$4 \text{ models} \times 11 \text{ tasks} \times 2 \text{ libraries} \times 3 \text{ runs} = 264. \tag{3}$$

All runs used temperature 0.0 and at most one repair attempt.

Table 2: Benchmark configuration.

Item	Value
Valid model outputs	`gpt-4o-mini`, `gpt-4.1-mini`, `gpt-5.4-mini`, `gpt-5.5`
Attempted but excluded	`gpt-5.3-codex` (API/harness failure)
Visualization libraries	Matplotlib, Altair
Tasks	11 initial visualization tasks
Runs per model-task-library condition	3
Valid generations	264
Temperature	0.0
Maximum repair attempts	1
Primary metric	First-pass total generation tokens
Secondary metrics	Code lines, execution success, repair tokens

6.3 Dataset and Schema

The experiment uses a synthetic tabular dataset of 5,000 rows generated with a fixed seed. The schema is designed to support common analytical visualization tasks while avoiding domain-specific interpretation. The dataset contains the fields `date`, `month`, `region`, `category`, `product`, `sales`, `revenue`, `margin`, `ad_spend`, `units`, `weekday`, and `hour`.

6.4 Prompt Construction

Each task is expressed as a natural-language visualization request. The model receives the same dataset schema and task description in both library conditions. The only intended difference between paired prompts is the required visualization library.

The prompt includes a fixed schema, the task text, the constraint to use either Matplotlib or Altair only, visualization quality requirements, and code constraints. The model is instructed to return executable Python code only, without markdown fences, explanations, comments, simulated data, or redefinition of the dataframe. These constraints are intended to reduce uncontrolled variation caused by explanatory text or example data.

6.5 Visualization Tasks

The benchmark includes tasks of increasing complexity, ranging from simple static charts to compound and lightly interactive visualizations.

The complexity classes are experimental strata assigned before analysis; they are not claimed to be objective measures of visualization complexity.

6.6 Execution and Repair Protocol

For each generated response, the returned code is extracted and executed in a controlled Python environment. If the generated code executes successfully, the number of repair attempts and repair-token

Table 3: Visualization tasks and manually assigned complexity classes.

Task	Chart type	Interaction	Complexity
T1	Bar chart	None	Low
T2	Line chart / time series	None	Low
T3	Scatter plot	Tooltip	Medium
T4	Histogram	None	Low
T5	Grouped bar chart	None	Medium
T6	Multi-line chart	None	Medium-high
T7	Faceted chart	None	Medium-high
T8	Heatmap	None	Medium
T9	Category selection	Selection + tooltip	High
T10	Dropdown filter	Parameter control	High
T11	Linked views	Linked selection	High

cost are recorded as zero. If execution fails, the failed code and the corresponding Python error message are returned to the model using a standardized repair prompt. The model is asked to fix the code while preserving the original task and required library. Each repair attempt is counted and its token cost is recorded separately.

6.7 Measured Quantities

For each run, the following quantities are recorded: model provider and model name; task identifier; library condition; run identifier; prompt tokens; completion tokens; total initial tokens; generated code length in characters and lines; first-pass execution success; number of repair attempts; repair prompt tokens; repair completion tokens; total repair tokens; final execution success; and final error message, if any.

The primary dependent variable is the total token cost of the first generation, T_{initial}. Code-line count is used as a secondary proxy for procedural burden. Repair tokens and execution success are reported as reliability metrics. Across repeated runs, repeat-run variability is also reported descriptively using the mean and standard deviation, $\mu \pm \sigma$, of first-pass tokens, generated code lines, and repair tokens within each model-complexity-library group.

6.8 Scope of Evaluation

The present evaluation focuses on token cost, generated code length, and execution success. It does not claim that executable visualizations are necessarily semantically correct. A chart may run successfully while still encoding an incorrect field, aggregation, grouping, or interaction. Explicit semantic-fidelity evaluation is left for future work.

7 Results

Table 4 reports the primary model-by-complexity results. The effect is complexity-dependent. For low-complexity tasks, Altair does not provide a consistent token advantage. For medium and higher-complexity tasks, Altair reduces first-pass tokens in most model conditions, with the strongest reductions appearing in the high-complexity group.

Table 5 reports descriptive repeat-run variability using $\mu \pm \sigma$ for first-pass tokens, generated code lines, and repair tokens. These statistics are not used as a primary outcome measure, but as a secondary diagnostic of repeat-run behavior. In several medium and high-complexity conditions, Altair exhibits lower first-pass token dispersion than Matplotlib, but the pattern is not universal. Repair-token variability

Table 4: Primary token and code-line results by model and task complexity. Token reduction is computed as $(\text{Matplotlib} - \text{Altair})/\text{Matplotlib}$. Positive values indicate lower cost for Altair. Bold values indicate reductions of at least 10%.

Model	Complexity	MPL tokens	Altair tokens	Token red. (%)	MPL lines	Altair lines	Line red. (%)
gpt-4o-mini	low	381.3	421.8	-10.6	11.6	16.0	-38.5
gpt-4o-mini	medium	470.1	435.6	7.4	19.9	16.4	**17.3**
gpt-4o-mini	medium-high	422.0	434.0	-2.8	16.5	19.2	-16.2
gpt-4o-mini	high	561.1	469.9	**16.3**	28.4	22.6	**20.7**
gpt-4.1-mini	low	376.7	375.9	0.2	10.3	11.4	-10.8
gpt-4.1-mini	medium	512.3	424.1	**17.2**	25.3	15.7	**38.2**
gpt-4.1-mini	medium-high	467.0	419.5	**10.2**	20.0	15.0	**25.0**
gpt-4.1-mini	high	663.2	472.2	**28.8**	42.9	22.8	**46.9**
gpt-5.4-mini	low	402.6	427.1	-6.1	12.0	16.9	-40.7
gpt-5.4-mini	medium	562.9	463.0	**17.7**	34.4	23.1	**32.9**
gpt-5.4-mini	medium-high	508.8	441.7	**13.2**	24.2	20.3	**15.9**
gpt-5.4-mini	high	856.0	572.9	**33.1**	62.7	33.7	**46.3**
gpt-5.5	low	636.9	755.8	-18.7	14.2	12.2	**14.1**
gpt-5.5	medium	1168.3	778.1	**33.4**	41.0	17.9	**56.4**
gpt-5.5	medium-high	1069.2	873.7	**18.3**	29.8	16.7	**44.1**
gpt-5.5	high	2098.3	1365.3	**34.9**	29.1	27.1	6.9

remains large in conditions where execution failures occur. Therefore, variability should be interpreted as descriptive evidence about repeated generations, not as a formal stability claim.

Table 6 aggregates results across the four successfully evaluated models. In aggregate, Altair is more expensive for low-complexity charts, but reduces first-pass tokens by 22.6% for medium tasks, 12.1% for medium-high tasks, and 31.1% for high-complexity tasks. Code-line reductions follow the same pattern, suggesting that the effect is associated with reduced procedural plotting burden rather than only shorter syntax.

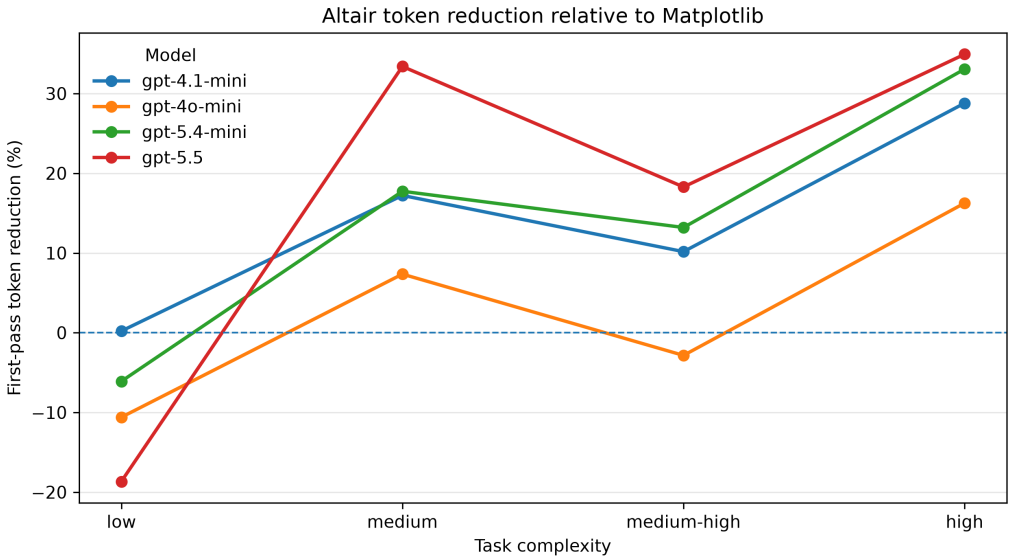

Figure 1: First-pass token reduction of Altair relative to Matplotlib by task complexity and model. Positive values indicate fewer generation tokens for Altair. Altair provides little or no consistent advantage for low-complexity charts, but reduces token cost for medium and high-complexity visualization tasks.

Figure 1 illustrates that the declarative advantage is model-dependent. The smallest model, `gpt-4o-mini`, exhibits weaker and less consistent gains, whereas the stronger models show clearer reductions for medium and high-complexity tasks. This supports the interpretation that declarative interfaces are most effective when the model can map task intent onto the semantic primitives of the visualization grammar.

Table 5: Descriptive repeat-run variability by model, task complexity, and visualization library. Values are reported as $\mu \pm \sigma$ for first-pass tokens, generated code lines, and repair tokens. Lower σ indicates lower observed dispersion across repeated generations, but no formal stability claim is made.

Model	Complexity	Library	Tokens ($\mu \pm \sigma$)	Lines ($\mu \pm \sigma$)	Repair tok. ($\mu \pm \sigma$)	n
gpt-4o-mini	low	Altair	421.8 ± 21.7	16.0 ± 0.9	48.2 ± 144.7	9
gpt-4o-mini	low	Matplotlib	381.3 ± 16.8	11.6 ± 1.2	0.0 ± 0.0	9
gpt-4o-mini	medium	Altair	435.6 ± 21.0	16.4 ± 1.9	0.0 ± 0.0	9
gpt-4o-mini	medium	Matplotlib	470.1 ± 85.4	19.9 ± 8.9	0.0 ± 0.0	9
gpt-4o-mini	medium-high	Altair	434.0 ± 7.3	19.2 ± 1.9	0.0 ± 0.0	6
gpt-4o-mini	medium-high	Matplotlib	422.0 ± 39.4	16.5 ± 3.8	0.0 ± 0.0	6
gpt-4o-mini	high	Altair	469.9 ± 14.2	22.6 ± 4.9	552.8 ± 418.3	9
gpt-4o-mini	high	Matplotlib	561.1 ± 73.3	28.4 ± 4.2	657.0 ± 543.1	9
gpt-4.1-mini	low	Altair	375.9 ± 15.0	11.4 ± 1.1	0.0 ± 0.0	9
gpt-4.1-mini	low	Matplotlib	376.7 ± 19.4	10.3 ± 1.0	0.0 ± 0.0	9
gpt-4.1-mini	medium	Altair	424.1 ± 37.7	15.7 ± 2.2	0.0 ± 0.0	9
gpt-4.1-mini	medium	Matplotlib	512.3 ± 62.3	25.3 ± 7.4	0.0 ± 0.0	9
gpt-4.1-mini	medium-high	Altair	419.5 ± 26.1	15.0 ± 1.1	0.0 ± 0.0	6
gpt-4.1-mini	medium-high	Matplotlib	467.0 ± 46.0	20.0 ± 3.3	0.0 ± 0.0	6
gpt-4.1-mini	high	Altair	472.2 ± 33.8	22.8 ± 5.5	359.7 ± 539.5	9
gpt-4.1-mini	high	Matplotlib	663.2 ± 111.0	42.9 ± 13.2	0.0 ± 0.0	9
gpt-5.4-mini	low	Altair	427.1 ± 24.6	16.9 ± 4.1	0.0 ± 0.0	9
gpt-5.4-mini	low	Matplotlib	402.6 ± 38.4	12.0 ± 2.7	0.0 ± 0.0	9
gpt-5.4-mini	medium	Altair	463.0 ± 21.6	23.1 ± 1.7	0.0 ± 0.0	9
gpt-5.4-mini	medium	Matplotlib	562.9 ± 97.2	34.4 ± 14.7	0.0 ± 0.0	9
gpt-5.4-mini	medium-high	Altair	441.7 ± 22.3	20.3 ± 3.3	0.0 ± 0.0	6
gpt-5.4-mini	medium-high	Matplotlib	508.8 ± 66.3	24.2 ± 5.7	0.0 ± 0.0	6
gpt-5.4-mini	high	Altair	572.9 ± 79.8	33.7 ± 8.4	119.8 ± 359.3	9
gpt-5.4-mini	high	Matplotlib	856.0 ± 163.9	62.7 ± 17.6	374.0 ± 592.8	9
gpt-5.5	low	Altair	755.8 ± 118.2	12.2 ± 1.2	76.1 ± 228.3	9
gpt-5.5	low	Matplotlib	636.9 ± 195.9	14.2 ± 4.3	0.0 ± 0.0	9
gpt-5.5	medium	Altair	778.1 ± 118.7	17.9 ± 2.1	130.9 ± 392.7	9
gpt-5.5	medium	Matplotlib	1168.3 ± 213.3	41.0 ± 7.1	0.0 ± 0.0	9
gpt-5.5	medium-high	Altair	873.7 ± 57.6	16.7 ± 2.9	0.0 ± 0.0	6
gpt-5.5	medium-high	Matplotlib	1069.2 ± 298.2	29.8 ± 6.3	0.0 ± 0.0	6
gpt-5.5	high	Altair	1365.3 ± 613.3	27.1 ± 11.1	0.0 ± 0.0	9
gpt-5.5	high	Matplotlib	2098.3 ± 333.0	29.1 ± 43.9	0.0 ± 0.0	9

Table 6: Aggregate results across the four successfully evaluated OpenAI models.

Complexity	MPL tokens	Altair tokens	Token red. (%)	MPL lines	Altair lines	Line red. (%)
low	449.4	495.1	-10.2	12.0	14.1	-17.6
medium	678.4	525.2	22.6	30.2	18.3	39.4
medium-high	616.8	542.2	12.1	22.6	17.8	21.4
high	1044.7	720.1	31.1	40.8	26.5	34.9

Table 7 reports execution robustness. Repair tokens are not included in the primary token-efficiency metric. They are reported separately because repair behavior depends on the execution environment, library-version compatibility, and the repair-loop implementation. The table also shows that repair behavior is not library-specific: for high-complexity tasks, both Matplotlib and Altair can require repairs depending on the model.

Table 7: Execution robustness and repair behavior. Success rates are percentages. Repair tokens are reported separately from the primary first-pass token metric. Bold repair-token values indicate non-zero repair cost; bold final-success values indicate final execution below 100%.

Model	Complexity	MPL first (%)	Altair first (%)	MPL final (%)	Altair final (%)	MPL repair tok.	Altair repair tok.
gpt-4o-mini	low	100.0	88.9	100.0	100.0	0.0	**48.2**
gpt-4o-mini	medium	100.0	100.0	100.0	100.0	0.0	0.0
gpt-4o-mini	medium-high	100.0	100.0	100.0	100.0	0.0	0.0
gpt-4o-mini	high	33.3	33.3	**77.8**	**77.8**	**657.0**	**552.8**
gpt-4.1-mini	low	100.0	100.0	100.0	100.0	0.0	0.0
gpt-4.1-mini	medium	100.0	100.0	100.0	100.0	0.0	0.0
gpt-4.1-mini	medium-high	100.0	100.0	100.0	100.0	0.0	0.0
gpt-4.1-mini	high	100.0	66.7	100.0	**66.7**	0.0	**359.7**
gpt-5.4-mini	low	100.0	100.0	100.0	100.0	0.0	0.0
gpt-5.4-mini	medium	100.0	100.0	100.0	100.0	0.0	0.0
gpt-5.4-mini	medium-high	100.0	100.0	100.0	100.0	0.0	0.0
gpt-5.4-mini	high	66.7	88.9	100.0	100.0	**374.0**	**119.8**
gpt-5.5	low	100.0	88.9	100.0	100.0	0.0	**76.1**
gpt-5.5	medium	100.0	88.9	100.0	100.0	0.0	**130.9**
gpt-5.5	medium-high	100.0	100.0	100.0	100.0	0.0	0.0
gpt-5.5	high	100.0	100.0	100.0	100.0	0.0	0.0

8 Discussion

The main empirical result is not that Altair is always shorter than Matplotlib. It is not. The result is more specific: the token advantage of Altair appears when the visualization task contains enough semantic structure for a declarative grammar to absorb part of the agent's procedural burden. For low-complexity charts, the agentic task is already close to the primitive plotting interface: create a figure, choose one mark, assign one or two axes, and add labels. In that regime, the overhead of a declarative grammar can equal or exceed its benefit. For medium and high-complexity tasks, however, the task structure increasingly involves aggregation, grouping, faceting, selections, filters, tooltips, and composition. These are precisely the operations that Altair/Vega-Lite exposes as semantic primitives. The reduction in first-pass tokens therefore becomes visible when the task moves from drawing a chart to specifying a visualization structure.

This distinction suggests that token efficiency should not be understood as a purely syntactic property. A library is not token-efficient simply because its programs contain fewer characters or fewer lines in isolation. Instead, token efficiency depends on the match between three structures: the semantic structure of the user request, the abstraction structure of the tool, and the model's ability to map the former into the latter. In this benchmark, Altair provides a compact target language for tasks that can be expressed as mappings from data fields to visual encodings, transformations, and interactions. When the model can recognize this mapping, it can express the visualization intent directly. When it cannot, the declarative interface may no longer provide an advantage and may even introduce additional tokens.

This provides a useful interpretation of the model-dependent results. The weaker and less consistent gains observed for `gpt-4o-mini` suggest that declarative interfaces do not automatically reduce token cost. A model must know how to use the declarative grammar as a semantic target language. More capable models appear to exploit this structure more consistently, especially for interactive and composed tasks. This is important for agent design: the benefit of a tool is conditional not only on the tool's formal expressiveness, but also on whether the agent can reliably plan in the abstraction space exposed by that

tool. A declarative grammar is therefore not merely an API. For an agent, it functions as an intermediate representation.

The Matplotlib condition illustrates the complementary case. Matplotlib is highly expressive and gives fine-grained control over figures, axes, artists, legends, scales, and rendering details. This flexibility is valuable for customized scientific figures and low-level visual control. However, in agentic generation, this flexibility also expands the action space. The agent must decide not only what the visualization means, but how to operationalize it procedurally. For grouped, faceted, interactive, or linked visualizations, this often requires loops, temporary aggregated dataframes, manual layout construction, legend management, and event or widget logic. These choices are not always part of the user's analytical intent; they are consequences of the interface. From the perspective of an agent, they are additional decisions that consume tokens and create opportunities for error.

Altair changes this burden by making common visualization semantics first-class. Encodings, marks, transformations, selections, facets, and compositions are represented directly in the grammar. This does not remove the need for semantic reasoning. The model must still infer which fields are relevant, which aggregations are appropriate, and which visual encodings match the task. Rather, Altair reduces the amount of procedural code required after the semantic decision has been made. In this sense, the declarative interface compresses the expression of an already-solved semantic problem. The relevant compression is not only textual compression, but representational compression: fewer low-level choices are needed to express the same visualization intent.

This distinction also explains why code-line reductions and token reductions move together in the aggregate results. If Altair were only shorter because of superficial syntax, one would expect weaker alignment between token reduction and code-line reduction. Instead, the reductions are strongest in the complexity strata where procedural plotting logic would otherwise expand. This supports the interpretation that the observed effect is associated with reduced procedural burden. The generated programs are not merely shorter; they are shorter because more of the task is delegated to the declarative semantics of the visualization grammar.

The repeat-run variability analysis provides a secondary, descriptive view of the same phenomenon. In several medium and high-complexity conditions, Altair has lower first-pass token dispersion than Matplotlib, which is consistent with the idea that declarative grammars can constrain the space of generated specifications. However, the evidence is descriptive and should not be interpreted as proving that Altair universally stabilizes generation. Variability remains model- and task-dependent, and repair events can dominate the observed dispersion. The safer interpretation is that declarative interfaces may reduce unnecessary variation when the task aligns well with the grammar, but this requires further evaluation with more repetitions and broader task families.

The repair results should be interpreted separately. Repair tokens are part of the operational cost of an agentic workflow, but they do not measure the same phenomenon as first-pass token cost. A repair may be caused by a library-version mismatch, an execution-environment issue, an API change, renderer behavior, or a model-specific failure mode. These factors matter for deployment, but they confound the measurement of representational efficiency. This is why the present experiment treats first-pass generation tokens as the primary measure and repair tokens as a reliability dimension. The distinction is important: a representation can reduce the initial action space while still being vulnerable to execution failures if the model uses outdated API constructs or if the environment does not support a generated interaction pattern.

At the same time, repair behavior is not irrelevant. In practical agentic systems, total cost includes both initial generation and correction. A useful interface must therefore support both concise specification and robust execution. The results show that neither Matplotlib nor Altair dominates uniformly in repair behavior. For some high-complexity tasks, both libraries required repairs depending on the model. This suggests that tool choice alone is insufficient. Robust agentic visualization requires a broader system design: version-aware prompting, constrained code generation, library-specific validators, execution feedback, and possibly intermediate representations that can be checked before code is emitted.

The broader implication is that agentic AI systems should not be optimized only by selecting smaller

models, increasing context windows, or compressing prompts. Those techniques address token cost at the surface level. The present results point to a different optimization axis: choosing interfaces whose abstractions align with the structure of the task. In visualization, this means using grammars that expose fields, encodings, aggregations, selections, and compositions directly. In other domains, the analogous design principle would be to expose task-relevant semantic operations rather than forcing the agent to synthesize low-level procedural steps. The cost reduction comes from shrinking the agent's effective action space.

This has consequences for enterprise AI architecture. Many enterprise agent workflows are currently designed around general-purpose code generation: the model receives a task, writes code against a broad library, executes it, observes errors, and repairs the result. This pattern is flexible, but it can be token-expensive and fragile. A semantic-interface approach suggests a different architecture. Instead of asking an agent to generate arbitrary procedural code, the system can provide constrained declarative target languages for common task families: visualization specifications, report schemas, query plans, document templates, transformation graphs, or workflow descriptions. The model's role becomes mapping user intent into a structured representation whose primitives already correspond to the domain. This can reduce cost, improve controllability, and make validation easier.

For visualization specifically, Altair/Vega-Lite is an example of such a target language. It is not merely a plotting library accessed through Python. It is a compact specification language for a large class of analytical graphics. This makes it especially suitable for agents because it externalizes many visualization decisions into a grammar that can be inspected, validated, modified, serialized, and reused. A Matplotlib program often encodes visualization intent implicitly in procedural steps. An Altair specification encodes much of that intent explicitly. This difference matters when agents must generate, modify, debug, and communicate visualizations over multiple turns.

The results also suggest a distinction between expressive power and agentic suitability. Matplotlib is more general in many respects, but generality is not always the best interface property for an agent. An agent benefits from abstractions that are neither too low-level nor too rigid. If the abstraction is too low-level, the model spends tokens reconstructing procedural machinery. If it is too high-level or poorly aligned, the model cannot express the task without workarounds. Altair appears useful in the middle: it is constrained enough to reduce procedural burden, but expressive enough to represent a broad class of statistical and interactive graphics.

This framing also clarifies the role of model capability. As models become stronger, one might expect procedural code generation to become less costly. The results do not fully support that intuition. Stronger models can generate more elaborate procedural code, but this may increase rather than decrease token consumption. The largest model in the benchmark often produced substantially longer Matplotlib outputs, especially for complex tasks. This suggests that capability can amplify verbosity when the target interface invites procedural elaboration. Declarative interfaces can channel that capability into more compact semantic specifications. In other words, stronger models do not eliminate the need for good interfaces; they may make interface design more important.

The study therefore supports a representation-centered view of agentic efficiency. Token cost is not only a property of the model or the prompt. It is also a property of the representational contract between the agent and the tools it uses. A well-designed tool interface acts as a compression layer between intent and execution. It allows the model to express high-level decisions without repeatedly reconstructing low-level implementation details. In visualization, Altair provides such a layer for many common analytical graphics. The evidence in this benchmark suggests that this layer becomes increasingly valuable as visualization tasks become more structured.

Several limitations temper this interpretation. The benchmark uses a synthetic dataset, a fixed set of tasks, manually assigned complexity classes, and a limited number of repeated runs. It measures token cost, code length, execution success, repair behavior, and descriptive repeat-run variability, but not semantic fidelity or human-perceived visualization quality. The results should therefore be interpreted as evidence for a controlled pattern, not as a universal ranking of visualization libraries. Future work should evaluate whether the same pattern holds across real-world datasets, broader visualization tasks, additional

declarative and imperative libraries, and explicit semantic-quality metrics.

Finally, the most important implication is conceptual. The objective is not to replace primitive-based plotting libraries, which remain essential for many forms of scientific and customized visualization. Rather, the objective is to identify when agentic systems should operate over semantic specifications instead of procedural primitives. The results indicate that when the task is naturally semantic, and when the tool exposes that semantics directly, declarative interfaces can reduce the cost of agentic generation. This makes Altair not only a visualization library, but an example of a broader design principle for agentic AI: use tools whose abstractions match the reasoning structure of the task.

9 Threats to Validity

Model dependence. Different language models may have different levels of familiarity with each visualization library and different abilities to exploit declarative abstractions. The results should therefore be interpreted as model-conditioned rather than universal.

API and token-accounting dependence. Token counts depend on the tokenizer, model API, response format, and experimental harness used at the time of the experiment. The absolute token values should therefore not be interpreted as stable constants across providers, model versions, or future API revisions. The main result is the relative comparison between paired Matplotlib and Altair conditions under the same protocol.

Prompt sensitivity. Small changes in prompt wording may affect generated code quality, verbosity, and token consumption. The experiment controls prompt structure across library conditions, but does not exhaustively test alternative prompt formulations.

Library familiarity bias. If a model was trained on more examples of one library than another, this may affect performance independently of the interface abstraction. The observed advantage may therefore reflect both the semantic structure of the interface and the model's learned familiarity with that interface.

Task selection bias. Altair may be especially strong for declarative statistical graphics, encodings, faceting, and interaction patterns that are naturally expressed in Vega-Lite. Matplotlib may be preferable for highly customized low-level rendering, specialized figure manipulation, or visualizations requiring imperative control.

Synthetic dataset. The benchmark uses a synthetic tabular dataset with a controlled schema. Real-world datasets may contain missing values, ambiguous column names, domain-specific semantics, inconsistent types, or preprocessing requirements. These factors may change both semantic reasoning cost and procedural plotting cost.

Manual complexity stratification. The task complexity classes are manually assigned. They are used as an experimental stratification, not as an objective complexity measure. A more formal complexity measure could consider the number of fields, transformations, encodings, views, interactions, and dependencies required by each task.

Limited repetitions. Each model-task-library condition was evaluated with three repeated runs. This is sufficient to reveal consistent directional patterns in this exploratory benchmark, but it is not intended to support strong statistical claims about variance, significance, or model-wide generalization.

Execution environment. Package availability, renderer behavior, and API-version compatibility can affect execution success and repair cost. This is why repair tokens are reported separately from first-pass generation tokens.

Semantic correctness. The present experiment measures generation cost and execution success, not human-perceived quality or semantic fidelity. A chart may run successfully while still encoding an incorrect field, aggregation, grouping, scale, or interaction. Explicit semantic-fidelity evaluation is left for future work.

10 Conclusion

This paper studied whether a declarative visualization grammar can reduce the token cost of agentic visualization generation. Using a controlled benchmark across 11 visualization tasks, two visualization libraries, three repeated runs, and four successfully evaluated OpenAI models, we compared Matplotlib as a primitive-based plotting interface with Altair/Vega-Lite as a declarative visualization interface.

The results support a complexity-dependent account of token efficiency. Altair does not consistently reduce first-pass generation tokens for low-complexity charts. However, for medium and high-complexity tasks, Altair reduces both first-pass generation tokens and generated code length. In aggregate, Altair reduces first-pass tokens by 22.6% for medium tasks, 12.1% for medium-high tasks, and 31.1% for high-complexity tasks. These reductions are accompanied by substantial code-line reductions, suggesting that the effect is not merely cosmetic syntax compression but a reduction in procedural plotting burden.

The descriptive variability results provide preliminary evidence that declarative specifications may reduce repeat-run dispersion in some medium and high-complexity conditions, although this effect is not universal and remains affected by model behavior and repair events.

The central implication is that token efficiency is not an intrinsic property of a library alone. It emerges from the alignment between the semantics of the user task, the semantic abstractions exposed by the tool, and the capability of the model to map one onto the other. Declarative visualization grammars are therefore most useful in agentic workflows when they expose abstractions that match the structure of the reasoning already performed by the model: fields, types, aggregations, encodings, facets, selections, filters, and composed views.

Repair behavior should be interpreted separately from first-pass generation cost. In this experiment, repair tokens were affected by execution failures, library-version compatibility, and model-specific repair behavior. They are therefore better understood as a reliability dimension than as the primary measure of representational efficiency. This distinction is important for the design of agentic systems: reducing the initial action space and reducing execution failures are related but not identical objectives.

Overall, the results suggest that declarative tools such as Altair can act as token-efficient interfaces for agentic AI, not because they are always shorter, but because they allow capable models to express visualization intent closer to its semantic form. Future work should extend this benchmark to semantic-fidelity evaluation, iterative modification tasks, additional visualization libraries, real-world datasets, and broader model families.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse,

Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[2] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[3] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2017.

[4] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Vega: A visualization grammar. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 2016.

[5] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2023.

[6] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. Altair: Interactive statistical visualizations for python. *Journal of Open Source Software*, 3(32):1057, 2018.

[7] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[8] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag, New York, 2016.

[9] Leland Wilkinson. *The Grammar of Graphics*. Statistics and Computing. Springer-Verlag, New York, 2 edition, 2005.

[10] Leland Wilkinson. The grammar of graphics. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(6):673–677, 2010.

[11] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

A Experimental Prompts

This appendix reports the task prompts used in the experimental evaluation. Each visualization task was executed under two conditions: a primitive-based plotting condition and a declarative Altair condition. The dataset schema was kept fixed across all prompts.

A.1 Initial Visualization Tasks

T1: Bar chart.

> Create a bar chart showing total revenue by product category. Sort the categories by descending total revenue and include a clear title and axis labels.

T2: Line chart.

> Create a line chart showing monthly total revenue over time. The x-axis should represent month and the y-axis should represent total revenue.

T3: Scatter plot with tooltip.

Create a scatter plot showing the relationship between advertising spend and revenue. Add a tooltip showing month, category, ad spend, and revenue.

T4: Histogram.

Create a histogram showing the distribution of revenue values.

T5: Grouped bar chart.

Create a grouped bar chart showing total revenue by product category and region.

T6: Layered line chart.

Create a line chart showing monthly revenue by product category. Each category should be shown as a different line.

T7: Faceted chart.

Create a faceted chart showing monthly revenue trends separately for each region.

T8: Heatmap.

Create a heatmap showing average revenue by weekday and hour of day.

T9: Interactive category selection.

Create an interactive scatter plot where selecting a product category highlights the corresponding points. Add tooltips showing category, revenue, and margin.

T10: Dropdown filter.

Create a line chart of monthly revenue with a dropdown selector that allows the user to choose the region.

T11: Linked views.

Create a linked visualization with a scatter plot and a bar chart. Selecting points in the scatter plot should update the bar chart to show totals for the selected subset.

B Task-Level Results

Table 8: Task-level first-pass token results. Positive token reduction indicates lower cost for Altair.

Model	Task	Task name	Complexity	MPL tokens	Altair tokens	Token red. (%)
gpt-4o-mini	T1	Bar chart	low	393.0	436.7	-11.1
gpt-4o-mini	T2	Line chart	low	392.0	435.7	-11.1
gpt-4o-mini	T4	Histogram	low	359.0	393.0	-9.5
gpt-4o-mini	T3	Scatter plot with tooltip	medium	580.3	421.0	27.5
gpt-4o-mini	T5	Grouped bar chart	medium	394.0	423.3	-7.4
gpt-4o-mini	T8	Heatmap	medium	436.0	462.3	-6.0
gpt-4o-mini	T6	Layered line chart	medium-high	386.0	436.7	-13.1
gpt-4o-mini	T7	Faceted chart	medium-high	458.0	431.3	5.8
gpt-4o-mini	T10	Dropdown filter	high	487.3	467.0	4.2

Model	Task	Task name	Complexity	MPL tokens	Altair tokens	Token red. (%)
gpt-4o-mini	T11	Linked views	high	554.0	487.3	12.0
gpt-4o-mini	T9	Interactive category selection	high	642.0	455.3	29.1
gpt-4.1-mini	T1	Bar chart	low	392.0	382.7	2.4
gpt-4.1-mini	T2	Line chart	low	387.0	386.0	0.3
gpt-4.1-mini	T4	Histogram	low	351.0	359.0	-2.3
gpt-4.1-mini	T3	Scatter plot with tooltip	medium	594.7	395.0	33.6
gpt-4.1-mini	T5	Grouped bar chart	medium	469.0	403.3	14.0
gpt-4.1-mini	T8	Heatmap	medium	473.3	474.0	-0.1
gpt-4.1-mini	T6	Layered line chart	medium-high	425.0	396.0	6.8
gpt-4.1-mini	T7	Faceted chart	medium-high	509.0	443.0	13.0
gpt-4.1-mini	T10	Dropdown filter	high	577.7	454.0	21.4
gpt-4.1-mini	T11	Linked views	high	609.3	517.0	15.2
gpt-4.1-mini	T9	Interactive category selection	high	802.7	445.7	44.5
gpt-5.4-mini	T1	Bar chart	low	414.0	446.3	-7.8
gpt-5.4-mini	T2	Line chart	low	438.0	433.3	1.1
gpt-5.4-mini	T4	Histogram	low	355.7	401.7	-12.9
gpt-5.4-mini	T3	Scatter plot with tooltip	medium	686.7	465.0	32.3
gpt-5.4-mini	T5	Grouped bar chart	medium	483.7	450.3	6.9
gpt-5.4-mini	T8	Heatmap	medium	518.3	473.7	8.6
gpt-5.4-mini	T6	Layered line chart	medium-high	459.0	443.3	3.4
gpt-5.4-mini	T7	Faceted chart	medium-high	558.7	440.0	21.2
gpt-5.4-mini	T10	Dropdown filter	high	646.7	552.3	14.6
gpt-5.4-mini	T11	Linked views	high	951.0	671.7	29.4
gpt-5.4-mini	T9	Interactive category selection	high	970.3	494.7	49.0
gpt-5.5	T1	Bar chart	low	574.3	677.7	-18.0
gpt-5.5	T2	Line chart	low	826.0	801.3	3.0
gpt-5.5	T4	Histogram	low	510.3	788.3	-54.5
gpt-5.5	T3	Scatter plot with tooltip	medium	1185.7	646.3	45.5
gpt-5.5	T5	Grouped bar chart	medium	1037.7	843.7	18.7
gpt-5.5	T8	Heatmap	medium	1281.7	844.3	34.1
gpt-5.5	T6	Layered line chart	medium-high	872.3	890.3	-2.1
gpt-5.5	T7	Faceted chart	medium-high	1266.0	857.0	32.3
gpt-5.5	T10	Dropdown filter	high	2297.0	1808.3	21.3
gpt-5.5	T11	Linked views	high	2309.0	1434.0	37.9
gpt-5.5	T9	Interactive category selection	high	1689.0	853.7	49.5

C Repair Prompt Template

When generated code failed during execution, the following repair template was used:

> The following Python visualization code failed during execution:
>
> `<FAILED_CODE>`
>
> The execution produced this error:
>
> `<ERROR_MESSAGE>`
>
> Fix the code while preserving the original visualization task and the required library, `<LIBRARY>`.
>
> Return executable Python code only. Do not include explanations.

D Planned Modification Benchmark

Iterative modification tasks were considered in the design but are not included in the empirical results reported here. Future work will evaluate follow-up requests such as adding tooltips, changing aggregation, adding dropdown filters, faceting by region, composing charts, and adding linked selections.