

UNIVERSIDADE FEDERAL FLUMINENSE

LUCAS ALEXANDRE SIQUEIRA DOS SANTOS

**Autark: A Serverless Toolkit for Prototyping
Urban Visual Analytics Systems**

NITERÓI

2026

LUCAS ALEXANDRE SIQUEIRA DOS SANTOS

Autark: A Serverless Toolkit for Prototyping Urban Visual Analytics Systems

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação

Orientador:

MARCOS DE OLIVEIRA LAGE FERREIRA

NITERÓI

2026

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

S237a Santos, Lucas Alexandre Siqueira dos
Autark : A Serverless Toolkit for Prototyping Urban Visual
Analytics Systems / Lucas Alexandre Siqueira dos Santos. -
2026.
58 f.

Orientador: Marcos de Oliveira Lage.
Dissertação (mestrado)-Universidade Federal Fluminense,
Instituto de Computação, Niterói, 2026.

1. Análise Visual Urbana. 2. Toolkit de Visualização. 3.
Bancos de Dados Espaciais. 4. Processamento em GPU. 5.
Produção intelectual. I. Lage, Marcos de Oliveira,
orientador. II. Universidade Federal Fluminense. Instituto de
Computação. III. Título.

CDD - XXX

LUCAS ALEXANDRE SIQUEIRA DOS SANTOS

Autark: A Serverless Toolkit for Prototyping Urban Visual Analytics Systems

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação

Aprovada em Agosto de 2026.

BANCA EXAMINADORA

Prof. Marcos de Oliveira Lage - Orientador, UFF

Prof. Nivan Roberto Ferreira Júnior, UFPE

Prof. Daniel Cardoso Moraes de Oliveira, UFF

Niterói

2026

Resumo

O desenvolvimento de sistemas visuais de análise (VA) tem sido tradicionalmente um processo trabalhoso, que equilibra metodologias de design com práticas complexas de engenharia de software para lidar com fluxos de dados heterogêneos. Apesar da riqueza da literatura de VA urbana, a área carece de um *toolkit* consolidado que encapsule os componentes fundamentais de gerenciamento de dados espaciais e processamento analítico em uma estrutura única e leve. Para preencher essa lacuna, este trabalho apresenta o Autark, um *toolkit* executado inteiramente no lado do cliente (*fully client-side*), sem qualquer componente de servidor, rodando por completo no navegador da web, projetado para a prototipagem rápida de sistemas de VA urbana com base em um modelo centrado no elemento geográfico (*feature-centric*). Nesse contexto, esta dissertação foca primordialmente na concepção, no desenvolvimento e na integração de dois módulos centrais dessa arquitetura: o gerenciador de dados espaciais e o mecanismo de processamento analítico. O primeiro módulo consiste em um banco de dados espacial no lado do cliente, compilado para WebAssembly, responsável por unificar a ingestão e a consulta de diversas fontes de dados urbanos. O segundo módulo compreende um mecanismo de computação paralela acelerado por WebGPU que permite a execução de alto desempenho de operações analíticas em larga escala. A metodologia de integração estabelece uma interface padronizada que conecta de forma fluida os dados processados por esses dois módulos às bibliotecas de visualização do *toolkit*. Os resultados da avaliação de desempenho demonstram que os módulos desenvolvidos alcançam um comportamento de escalonamento linear e tempos de resposta interativos para o carregamento e processamento de dados espaciais inteiramente no lado do cliente. Além disso, em um experimento no qual agentes de IA geraram aplicações de VA urbana, comparamos o Autark com uma abordagem livre (*baseline*) e com o UTK, o *toolkit* de VA urbana mais próximo. As aplicações geradas com o Autark exibiram uma redução significativa na complexidade estrutural, com uma média de 222 linhas de código e complexidade ciclomática de 27, em comparação a 787 linhas e complexidade de 114 na abordagem base, ao mesmo tempo em que demandaram um esforço de desenvolvimento (em tempo e custo) substancialmente menor que o do UTK. Em conclusão, os componentes arquiteturais desenvolvidos minimizam efetivamente a sobrecarga

de infraestrutura associada à construção de protótipos de VA urbana, viabilizando análises interativas de alto desempenho e aprimorando substancialmente a confiabilidade do código em fluxos de trabalho de desenvolvimento assistidos por IA.

Palavras-chave: Análise Visual Urbana, Análise Urbana, Dados Urbanos, *Toolkit* de Visualização, Bancos de Dados Espaciais, Processamento em GPU.

Abstract

The development of visual analytics (VA) systems has traditionally been a labor-intensive process, balancing design methodologies with complex software engineering practices to handle heterogeneous data streams. Despite the richness of the urban VA literature, the field lacks a consolidated toolkit that encapsulates the core components of spatial data management and analytical processing into a unified, lightweight framework. To address this gap, this work introduces Autark, a fully client-side toolkit executed entirely in the web browser, with no backend infrastructure, designed for the rapid prototyping of urban VA systems based on a feature-centric model. Within this context, this dissertation focuses primarily on the design, development, and integration of two core modules of this architecture: the spatial data manager and the analytical processing engine. The first module consists of a client-side spatial database, compiled to WebAssembly, responsible for unifying the ingestion and querying of diverse urban data sources. The second module comprises a WebGPU-accelerated parallel computation engine that enables the high-performance execution of large-scale analytical operations. The integration methodology establishes a standardized interface that seamlessly connects the data processed by these two modules to the toolkit’s visualization libraries. Performance evaluation results demonstrate that the developed modules achieve linear scaling behavior and interactive response times for loading and processing spatial data entirely on the client side. Furthermore, in an experiment in which AI agents generated urban VA applications, we compared Autark against an unconstrained baseline and against UTK, the most closely related urban VA toolkit. Applications generated with Autark exhibited a significant reduction in structural complexity, averaging 222 lines of code and a cyclomatic complexity of 27, compared to 787 lines and a complexity of 114 in the baseline condition, while requiring development effort (in time and cost) substantially lower than UTK. In conclusion, the developed architectural components effectively dissolve the infrastructure overhead associated with building urban VA prototypes, empowering high-performance interactive analysis and substantially improving code reliability in AI-assisted development workflows.

Keywords: Urban Visual Analytics, Urban Analytics, Urban Data, Visualization Toolkit, Spatial Databases, GPU Processing.

Contents

1	Introduction	12
1.1	Motivation and Context	12
1.2	Problem Statement	13
1.3	Objectives	13
1.4	Contributions	14
1.5	Organization of the Dissertation	14
2	Related Work	16
2.1	Urban Visual Analytics Systems	16
2.2	Tools for Urban VA Development	16
2.3	The Urban Toolkit (UTK)	17
2.4	LLMs in Visual Analytics	19
3	Design Principles	20
3.1	Lessons from urban VA development landscape	20
3.2	Lessons from agentic VA development	21
4	The Autark Toolkit	24
4.1	A unified model for data and interaction in urban VA	24
4.2	The Autark components	26
4.2.1	Spatial database	26
4.2.2	GPU compute engine	30
4.2.3	3D map visualization	32

4.2.4	Abstract charts	34
5	Evaluation	37
5.1	Usage examples	37
5.2	Performance	42
5.2.1	Agentic urban VA development	42
6	Conclusion	51
	REFERENCES	54

1 Introduction

1.1 Motivation and Context

The creation of visual analytics (VA) systems is at the cornerstone of visualization as a field. Over the past decades, hundreds of systems have been proposed tackling domains, data, and problems as diverse as biological pathways (Partl *et al.*, 2013) and transportation routing (Guo *et al.*, 2011). Given that one of the main objectives is a software artifact, this creation process is not only guided by design methodologies (Munzner, 2009; Sedlmair; Meyer; Munzner, 2012), but also heavily influenced by software engineering practices and tools.

This software engineering dimension has been shaped by the tools available to visualization researchers. Visualization libraries (Bostock; Ogievetsky; Heer, 2011), grammars (Satyanarayan *et al.*, 2016), knowledge bases (Moritz *et al.*, 2018), and collaborative canvases (Bostock; Conlen *et al.*, 2024) have each redefined what it means to *prototype* a VA system: how fast an idea can be instantiated, validated with users, and discarded or refined. Today, we are in the midst of what may be the most disruptive such shift yet: the rise of large language models (LLMs) and AI-assisted coding.

Tools and services have fundamentally altered the programmer’s autonomy, compressing the distance between a design intention and a working implementation. AI-assisted coding qualitatively changes what a single researcher can sustain alone; components that once required hours of careful engineering can now be generated and adapted much faster, shifting the bottleneck from *writing code* to *specifying intent*. Yet, this new generative capability also surfaces new structural challenges, as LLMs produce more reliable code when composing from well-defined, reusable abstractions rather than generating complex solutions from scratch (Stengel-Eskin; Prasad; Bansal, 2024). A promising strategy is to then provide well-scoped software libraries that encapsulate complex low-level implementations into high-level abstractions that can be reused to build systems.

1.2 Problem Statement

The visualization community has long proposed general and reusable toolkits (Bostock; Ogievetsky; Heer, 2011; Heer; Card; Landay, 2005), frameworks (Bostock; Heer, 2009), and component libraries (Wongsuphasawat, 2020). These general libraries, however expressive, leave a gap for domain-specific tasks, as their data types, analytical pipelines, and visualization encodings fall outside any general approach.

Urban VA is a particularly salient case. Cities generate heterogeneous data streams from trajectories, point-of-interest distributions, real-time sensor feeds, etc., each carrying its own spatial scale, temporal resolution, and analytical constraints (Bonadia *et al.*, 2023). Building VA systems on such data requires not only specific rendering capabilities but also a coherent set of domain abstractions. Despite the richness of the urban VA literature (Ferreira; Moreira; Hosseini *et al.*, 2024) and several targeted contributions (Miranda; Ortner *et al.*, 2024), the field lacks a consolidated, reusable toolkit for rapid prototyping, leaving researchers to reassemble the same foundational components with each new project. This often leads to complex architectures where multiple disconnected components handle data ingestion instead of lightweight, unified architectures.

1.3 Objectives

In this dissertation, we present **Autark**, a fully client-side toolkit¹ for rapid prototyping of urban visual analytics systems. Autark integrates the core components of urban visual analytics systems into a high-level, structured, and unified codebase. By consolidating essential components, such as spatial data management, parallel computing, and multi-scale visualization into a cohesive framework, Autark simplifies prototyping and experimentation.

Furthermore, we explore how Autark’s structured, tightly scoped interfaces leverage AI-assisted workflows. We hypothesize that by narrowing the design space and providing domain-aware abstractions, LLMs can achieve higher levels of code correctness. This integration allows developers to generate more reliable, higher-quality code, shifting the effort toward defining analytical goals and interaction logic. In doing so, we argue that a programmatic API over standard representations is a more effective natural-language

¹We use the term *fully client-side* rather than *serverless* deliberately. In cloud computing, *serverless* denotes managed server-side execution (Function-as-a-Service), in which servers still exist but are abstracted away from the developer. Autark requires no backend at all: data ingestion, querying, computation, and rendering all happen in the user’s browser, and deployment reduces to serving static files.

target for code generation than a custom declarative grammar coupled to a preprocessing backend, a thesis we develop against UTK throughout the dissertation.

1.4 Contributions

The technical core of this work includes the implementation of the **Spatial Database** and the **GPU Compute Engine** modules. The Spatial Database provides in-browser relational storage and querying powered by DuckDB compiled to WebAssembly. Complementing this, the GPU Compute Engine abstracts low-level WebGPU configuration details, providing a higher-level interface for data binding and shader definition. Together, these modules create a performant pipeline that ensures the feature abstraction remains unbroken.

The main contributions of this dissertation are:

1. The **Autark toolkit**, a fully client-side, backend-free architecture for rapid prototyping of urban VA, with emphasis on the abstraction of spatial data management and GPU-accelerated computation;
2. A **comparative study** of LLM coding effectiveness across three conditions (Autark, an unconstrained baseline, and UTK) evaluating both the structural quality of the agent-generated code and the agentic effort, in iterations, time, and cost, required to reach a working application;
3. A **series of usage scenarios** demonstrating its capability to streamline the creation of robust, shareable urban VA prototypes.

1.5 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 surveys the related work across three dimensions: urban VA systems and the architectural complexity they entail; the tools and libraries available for their development, including an in-depth comparison to UTK, the most closely related urban VA toolkit; and the emerging role of LLMs in visual analytics workflows. Chapter 3 distills the lessons learned from both the urban VA development landscape and our experience with agentic development workflows, establishing the four design principles that guided Autark’s design: Urban-specific, Web-first, Component-oriented, and LLM-ready. Chapter 4 presents the Autark toolkit in

detail. We introduce the feature-centric model, the unified programming abstraction that governs all inter-component communication, and then describe each of the four modules: the spatial database, the GPU compute engine, the 3D map viewer, and the abstract charts library. Chapter 5 evaluates Autark along three dimensions: a series of usage scenarios showing that complex urban VA systems can be prototyped with substantially less code; performance benchmarks assessing the scalability of the browser-based architecture; and a controlled experiment comparing agent-generated implementations across three conditions (Autark, an unconstrained baseline, and UTK) along two axes: the structural quality of the generated code and the agentic effort, in iterations, time, and cost, required to reach a working application. Chapter 6 concludes the dissertation with a discussion of community adoption, the inherent limitations of a fully client-side approach, and directions for future work.

2 Related Work

2.1 Urban Visual Analytics Systems

Urban VA systems have been developed across a wide range of data domains, including human mobility ([Andrienko et al., 2017](#)), noise pollution ([Rulff et al., 2022](#)), environmental simulations ([Deng et al., 2019](#)), public safety ([García-Zanabria et al., 2021](#)), and energy consumption ([Rodgers; Bartram, 2011](#)). Despite this diversity, these systems share a common requirement for a non-trivial combination of sophisticated modules responsible for data management, interactive visualization, and analytical processing.

TaxiVis ([Ferreira; Poco et al., 2013](#)), a foundational work in this field, exemplifies this complexity. To enable the interactive exploration of massive taxi trip datasets, the system requires a custom storage manager for efficient spatiotemporal retrieval, a visual query module, and multiple linked views. These systems become even more complex when three-dimensional data is required. **Urbane** ([Ferreira; Lage et al., 2015](#)), for instance, integrates 2D and 3D data layers for tasks such as impact analysis and landmark visibility.

A comprehensive survey by Miranda et al. ([Miranda; Ortner et al., 2024](#)) reveals that 3D urban systems must address challenges such as occlusion from dense geometry ([Elmqvist; Tsigas, 2008](#)) and navigation across spatial scales ([Chen et al., 2021](#)). These demands compound the integration burden, requiring specialized modules that increase development effort. Recent studies, such as the **VA-Blueprint** methodology ([Ferreira; Moreira; Miranda, 2025](#)), empirically demonstrate this: a typical urban VA system comprises approximately 25 distinct components, a number that has been growing over time.

2.2 Tools for Urban VA Development

Traditionally, GIS platforms like **ArcGIS** ([Booth; Mitchell et al., 2001](#)) and **QGIS** ([Moyroud; Portet, 2018](#)) have been the default tools for urban analysis. However, their mo-

nolithic design often hinders the rapid prototyping cycles required in visualization research (Ziegler; Chasins, 2023).

More recently, web-based libraries such as **Deck.gl** (DECK..., 2024), **Mapbox** (MAPBOX GL JS..., 2024), and **MapLibre** (MAPLIBRE GL JS..., 2024) have offered modular building blocks for spatial rendering. General-purpose grammars like **Vega-lite** (Satyanarayan *et al.*, 2016) further enable the declarative specification of visualizations. Specifically tailored to urban use cases, the **Urban Toolkit (UTK)** (Moreira; Hosseini; Nipu *et al.*, 2023) proposes a framework that integrates thematic and physical data layers across multiple scales.

Interactive urban VA also demands efficient data management. Early systems like **imMens** (Liu; Jiang; Heer, 2013) and **Nanocubes** (Lins; Klosowski; Scheidegger, 2013) introduced precomputed structures for scalability. Modern architectures like **Mosaic** (Heer; Moritz; Pechuk, 2025) leverage **DuckDB** (Raasveldt; Mühleisen, 2019) to provide low-latency cross-filtering. While these tools are powerful, they are often agnostic to urban-specific operations, such as spatial joins between thematic data and building footprints. **Autark** addresses these gaps by consolidating spatial management, GPU computation, and 3D visualization into a single fully client-side toolkit.

2.3 The Urban Toolkit (UTK)

The **Urban Toolkit (UTK)** (Moreira; Hosseini; Nipu *et al.*, 2023) is the work most closely related to Autark. It is a grammar-based framework for authoring web-based urban visualizations, built around three goals it shares with us, extensibility, reproducibility, and flexibility, and motivated by the same observation that urban VA systems are repeatedly rebuilt from scratch. UTK and Autark, however, pursue this goal from opposite ends: UTK offers a *declarative grammar* executed by a *client-server* runtime, whereas Autark offers a *programmatic, feature-centric* set of *client-only* components.

At the core of UTK is a JSON grammar organized around the concept of a **knot**: a declarative link that integrates a *thematic* layer (measurements over coordinates, e.g., sunlight or noise) with a *physical* layer (geometry such as buildings or streets) through a spatial relation and an aggregation. Knots can be chained across spatial scales and combined algebraically, which is how UTK expresses multi-scale aggregation and what-if scenario analysis; plots are authored as embedded Vega-Lite (Satyanarayan *et al.*, 2016) specifications. This grammar is UTK’s main strength: it is concise, reproducible (a

visualization is a shareable JSON file), and accessible enough that an urban expert can author 3D visualizations without writing rendering or graphics code.

That strength, however, comes with two structural costs that motivated different choices in Autark. First, UTK is largely static. Its grammar describes a *precomputed* result: data must first be downloaded, parsed, joined, and persisted as `.utk` files by a Python backend running in notebooks, and only then can the frontend render the corresponding grammar specification. Loading new data, changing a join, or introducing a layer means returning to the backend and re-running the preprocessing step. Autark, by contrast, is dynamic: ingestion, querying, and computation all happen at runtime in the browser, spatial and OLAP queries over DuckDB compiled to WebAssembly ([Raasveldt; Mühleisen, 2019](#)), and analytical computation on a general-purpose WebGPU engine. Because there is no ahead-of-time preprocessing barrier, an Autark application can adapt to inputs *as it runs*: a developer can change the code, or even build an application that reacts to end-user input and reconfigures its data and visualizations on the fly, a workflow that UTK’s preprocess-then-render pipeline does not naturally support.

Second, UTK relies on a proprietary data and specification format, the knot abstraction and the `.utk` files produced by its backend. Autark instead adopts the *geographical feature* as its atomic unit, the same representation used by OpenStreetMap and GeoJSON and effectively a community-wide standard for spatial data. In Autark every module exchanges collections of features, so data flows from ingestion to computation to rendering to interaction without ever crossing a proprietary boundary or requiring format conversion. This lowers the cost of bringing in external data and of reusing existing tooling, whereas UTK’s pipeline assumes data is first translated into its own format. These differences are felt even in a minimal “load and show an area” task: in UTK it spans two artifacts and a running server, a Python preprocessing step that persists the data and a user-authored JSON grammar that drives the rendering, whereas in Autark the equivalent is a single TypeScript fragment that ingests and renders in the same in-browser runtime, with no backend and no intermediate format.

In summary, UTK is best understood as a low-code authoring environment: its declarative grammar is an excellent fit for urban experts who want to compose and share 3D visualizations. Autark deliberately occupies the opposite corner of the design space: a fully in-browser toolkit that is trivial to deploy and share, dynamic rather than pre-computed, and built on a standard feature representation rather than a custom format. A central part of UTK’s rationale was accessibility: at the time it was proposed, a de-

clarative grammar was one of the few practical ways to let non-developers author urban visualizations without writing rendering or graphics code, and this was an explicit and well-justified motivation for adopting a domain-specific language. The recent maturation of LLM-assisted coding, which did not exist when UTK was designed, reframes this trade-off. Natural language can now drive a programmatic library directly, and a toolkit that exposes a clear, well-scoped interface over standard representations (such as Autark’s feature-centric API) is arguably an easier target for code generation than a bespoke grammar and data format. In other words, the ease of implementation that once justified a specialized grammar can increasingly be obtained from a general library through natural language, without committing users to a domain-specific language. We explore this possibility concretely in later chapters, where we use LLMs to generate urban VA applications from natural-language descriptions and compare both paradigms. Autark is therefore not a replacement for UTK, but an exploration of a different region of the same design space, one that the current generation of AI-assisted development makes newly attractive.

2.4 LLMs in Visual Analytics

Enabling users to author VA modules through natural language has long been a goal in the community. Early efforts like **DataTone** (Gao *et al.*, 2015) and **Flowsense** (Yu; Silva, 2019) relied on rule-based NLP or semantic parsing, but were limited to predefined visualization types.

The emergence of LLMs has dramatically expanded these capabilities. Systems such as **Urbanite** (Moreira; Ferreira *et al.*, 2026), **Chat2VIS** (Maddigan; Susnjak, 2023), and **LIDA** (Dibia, 2023) demonstrate that LLMs can automate visualization generation from free-form descriptions. This extends to 3D workflows with **ChatVis** (Mallick *et al.*, 2024) and real-time volumetric editing with **NLI4VolVis** (Ai; Tang; Wang, 2025).

Beyond individual artifacts, researchers are now leveraging LLMs to orchestrate entire systems. **LightVA** (Zhao *et al.*, 2024) decomposes analytical goals into subtasks, while **Data-to-Dashboard** (Zhang; Elhamod, 2025) uses modular agents to automate end-to-end pipelines. However, a fully general framework for any domain remains impractical. **Autark** fills this gap as a domain-specific tool, providing a structured framework that narrows the design space for LLMs, resulting in more robust and uniform urban VA systems.

3 Design Principles

3.1 Lessons from urban VA development landscape

Building VA systems for urban data presents multifaceted challenges arising from the intrinsic complexity of urban datasets, the architectural demands of such systems, the heterogeneous nature of their development ecosystem, and the high barriers to collaboration. All of these challenges shaped the design of **Autark**. Cities generate heterogeneous data streams, trajectories, sensor feeds, and environmental simulations, each carrying its own spatial scale, temporal resolution, and format, often reaching massive volumes (Bonadia *et al.*, 2023). Despite the richness of the urban VA literature (Miranda; Ortner *et al.*, 2024; Ferreira; Moreira; Hosseini *et al.*, 2024), the field lacks a consolidated toolkit for rapid prototyping, leaving researchers to reassemble the same foundational components with each new project.

General-purpose visualization libraries (Bostock; Ogievetsky; Heer, 2011; Satyanarayan *et al.*, 2016), however expressive, leave a gap for domain-specific tasks whose data types, analytical pipelines, and visualization encodings fall outside any general approach. This gap motivates a design that must target the recurring data types, spatial scales, and analytical patterns that characterize urban VA (**Urban-specific**), providing self-contained components that encode domain operations, such as spatial operations across multivariate data types, such as physical and thematic data, rather than requiring researchers to assemble these primitives from scratch.

From an architectural viewpoint, state-of-the-art urban visual analytics platforms typically integrate distinct modules for data ingestion and management, scalable computation and analytics, and both spatial and abstract visualizations (Doraiswamy *et al.*, 2018; Miranda; Ortner *et al.*, 2024). The orchestration of these modules may face different challenges depending on the adopted system architecture (e.g., client-server, micro-services, etc.). For example, client-server architectures, widely used in web application development, must address challenges such as data transfer latency, state synchronization

across distributed nodes, real-time performance during user interactions, and end-to-end security, particularly for sensitive urban datasets subject to privacy regulations.

These complex development infrastructures directly impact the practical results of urban VA systems. System deployment often becomes a difficult task that requires containerization and other specialized technologies, making tools hard to maintain and use. A toolkit designed to dissolve this overhead must propose a uniform architecture that facilitates distribution and reproducibility without requiring complex communication between heterogeneous services with different infrastructure requirements (**Web-first**). Autark’s fully client-side architecture is directly inspired by this observation. Because there is no backend service to orchestrate or deploy, none of these concerns arise. Its web-first stack, relying on **WebAssembly** and **WebGPU**, can be deployed as a static collection of modules, making systems inherently shareable and reproducible without environment configuration.

The design process of VA systems benefits from the ability to isolate and evaluate individual components independently, much like ablation studies. Replacing a spatial join strategy, swapping a rendering layer, or testing an alternative interaction model should not require restructuring the entire system. In practice, however, the tightly coupled architectures common in urban VA make such targeted experimentation difficult, discouraging exploration and creating barriers to collaboration with domain experts (Moreira; Hosseini; Veiga *et al.*, 2025), who cannot easily test ideas on isolated parts of the system. This motivates a design structured as a collection of loosely coupled, composable components with well-defined responsibilities and consistent interfaces (**Component-oriented**). Researchers can adopt individual components without committing to the full toolkit, and swap them as requirements evolve without cascading changes.

3.2 Lessons from agentic VA development

Employing LLMs in the construction of urban VA systems has the potential to mitigate the challenges previously outlined, particularly by accelerating prototyping, bridging expertise gaps, and streamlining the use of heterogeneous development ecosystems. In fact, LLMs can efficiently generate boilerplate code, support API definition, improve code quality, and aid debugging, among other benefits.

However, to fully exploit these benefits requires thorough context engineering, the deliberate curation of instructions that explicitly define the LLM’s expected outputs,

constraints, and success criteria. Calibrating the volume of information provided in the context is fundamental, as concise contexts yield outputs that fail to meet user goals, while large ones induce hallucinations due to diluted attention to key details.

In the context of urban visual analytics systems, one possible approach is to include general-purpose libraries useful for building these systems (e.g., spatial indexing libraries such as **DuckDB**, rendering frameworks such as **Three.js**, visualization libraries such as **D3.js**, or analytics tools such as **scikit-learn**) within the LLM.

However, integrating several low-level libraries into the LLM context significantly increases token consumption and operational costs. This approach often results in a large, generic context that fails to capture domain-specific requirements or optimize code structures, ultimately reducing the LLM’s effectiveness while increasing system maintenance costs.

Thus, the Autark’s APIs were designed to be structured and narrow, reducing the surface area an LLM must reason over (**LLM-ready**). Each component is documented with explicit contracts that can be included directly in a prompt context, benefiting both AI agents and human developers who must inspect and maintain code they did not write.

Reflecting on our experience designing urban VA systems (Ferreira; Poco *et al.*, 2013; Ferreira; Lage *et al.*, 2015; Miranda; Doraiswamy *et al.*, 2019; Moreira; Hosseini; Veiga *et al.*, 2025; Moreira; Hosseini; Nipu *et al.*, 2023; Rulff *et al.*, 2022) and surveying the field (Ferreira; Moreira; Hosseini *et al.*, 2024; Miranda; Ortner *et al.*, 2024), we observe that the ambition of a design idea is rarely the bottleneck; the operational overhead of realizing it is. With **Autark**, our philosophy is that a toolkit should dissolve this overhead rather than manage it, allowing researchers to focus entirely on the design problem at hand.

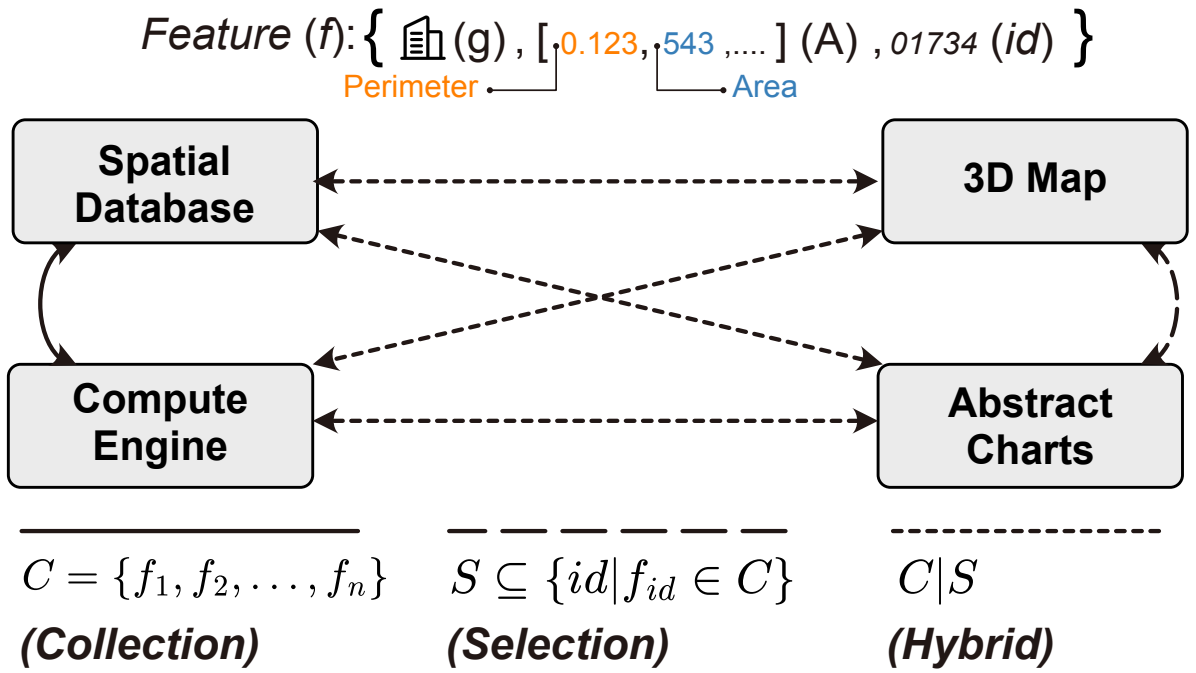


Figure 1: Autark’s fully client-side architecture. The toolkit comprises four independent modules that can be freely combined via their APIs. It uses feature collections C and selections S in accordance with the feature-centric model. The image shows the data that flows through each connection. Hybrid connections exchange either collections or selections, depending on the direction of flow.

4 The Autark Toolkit

Autark is a toolkit designed to streamline the prototyping of urban VA systems through a fully client-side architecture that executes entirely in the web browser, with no server-side component. What makes its modules composable, both for human developers and for agentic workflows, is a programming model that governs how all components communicate. We first describe this model, as it provides the conceptual foundation for understanding each module’s design. We then present Autark’s components, illustrated in Figure 1: a spatial database that provides in-browser relational storage and querying of heterogeneous urban datasets built over DuckDB; a GPU compute engine built using WebGPU that executes analytical operations in parallel across geographic features; a 3D map view that renders vector and raster spatial data also using WebGPU; and an abstract charts library that provides visualizations built using D3.js.

4.1 A unified model for data and interaction in urban VA

Urban VA systems require two features to work in tandem: a coherent strategy for integrating heterogeneous data streams across spatial scales and formats, and a mechanism for coordinating interactions across multiple linked views. Both problems in the urban setting can naturally be reduced to a common solution: geographical features describing the urban environment. Urban data, regardless of its origin, can be expressed as a collection of features with associated properties, and user interactions, regardless of their visual form, can be expressed as selections over those same features. Autark takes this observation as the foundation of its architecture model, making the geographical feature the atomic unit of the entire toolkit.

In most urban VA toolkits, data, rendering, and interaction each operate at different levels of abstraction: a database returns rows, a renderer consumes buffers, and an interaction produces a pixel coordinate or a bounding box. Autark collapses these into a single level: everything operates on features. In Autark, a spatial database ingests hete-

ogeneous urban data and exports it as a feature collection; a compute engine executes analytical operations across features and writes results back as new feature properties; a map component renders each feature’s geometry; and a chart component binds to feature properties. In this model, the developer’s effort to convert, reshape, or transfer data between components is minimized; instead, they reason about features and feature collections, and the toolkit ensures that every component understands the data.

We define this design as the **feature-centric model for urban VA**. This model’s central claim is that the complexity of urban VA systems often arises from a single architectural mismatch: data and interaction operate at different levels of abstraction. This mismatch is visible in the most influential VA programming models. In Vega-Lite, data is a flat array of tuples and interaction is expressed as a selection predicate that filters those tuples, a clean model for statistical visualization, but one that has no notion of geographic identity. In Mosaic, interaction is expressed as filtering predicates that translate user actions into SQL query constraints, but spatial selections must be expressed as a geometric predicate rather than as a reference to the features themselves. `deck.gl` moves closer to a feature-centric model by organizing data into layers that can operate directly on features. However, this consistency does not extend across the full system: the base map in `deck.gl` is a tile layer that has no notion of features, so a building in a layer and the geometry underneath it are fundamentally two different kinds of objects. In each case, the same translation layer reappears: between the data representation and the geographical entity, between the interaction event and the data it refers to, or both.

Autark avoids this by treating all data, including coastlines, streets, and buildings, as the same collection of features that drive computation and interaction. There is no base map, no tile layer, and no boundary at which the feature abstraction breaks down: the same primitive flows from data ingestion to rendering to interaction. In Autark, a feature $f = (g, A, id)$ is a triple consisting of a geometry g , a set of named thematic attributes $A = \{a_1, a_2, \dots, a_n\}$ and a unique identifier id . For example, a building is a feature whose geometry g is a polygon footprint, and whose attributes A may include height and year that it was built. A collection $C = \{f_1, f_2, \dots, f_n\}$ is a set of features that serves as the unit of exchange between all modules. A selection $S \subseteq \{id | f_{id} \in C\}$ is a subset of feature identifiers drawn from a collection. The model then supports two classes of operations, both defined over these primitives. A data operation $\delta : C \rightarrow C$ takes a collection and returns a collection that may be filtered, enriched, aggregated or transformed, but keeping the representation of features as the unit of output. An interaction operation $i : C \times E \rightarrow S$ takes a collection C and a user event E (e.g., a click or a brush gesture)

and returns a selection, i.e., the subset of feature identifiers S from C targeted by the user. Because both operations are defined over the same primitives, no translation is needed: a selection S produced by an interaction in one view can be applied directly to any collection C in any other view that shares the same identifier space.

This model offers three concrete advantages. First, it is semantically natural: a selected neighborhood is always a neighborhood, not an index into a buffer, not a row identifier in a result set, not a bounding box in pixel space, and developers reason about the same objects at every stage of the pipeline, from querying to computing to rendering to interacting. Second, it is composable: components can be freely combined because they share not just a data format but a common vocabulary for what the data means. Third, it aligns naturally with how urban data is sourced in practice: the two most common sources of urban physical layer data, OpenStreetMap and GeoJSON, both organize their content as collections of geographical features. This uniformity extends to non-spatial sources: when tabular data is ingested, Autark can join it to existing spatial features, so that all modules operate on the same feature-centric representation regardless of the original format. Next, we describe how this model is operationalized in practice using Autark’s components, each of which implements a subset of operations over a shared feature space.

4.2 The Autark components

Autark employs a fully client-side architecture, written in TypeScript, that executes entirely in the browser, eliminating backend dependencies and making deployment and distribution straightforward. As illustrated in Figure 1, Autark is composed of four core modules: a **spatial database** (Section 4.2.1), a general-purpose GPU **compute engine** (Section 4.2.2), a **3D map** viewer (Section 4.2.3), and a collection of **abstract charts** (Section 4.2.4). All modules are designed as standalone components and can be composed to handle complex usage scenarios. No centralized broker is required to orchestrate data exchanges between components. Components communicate via feature collections, the common communication protocol in Autark.

4.2.1 Spatial database

The spatial database module is the entry point of the feature-centric model. It provides in-browser relational storage and querying powered by DuckDB (compiled to WebAssembly), ingests heterogeneous urban datasets, and exports them as a collection of features C ,

the shared primitive that all downstream modules operate on. In terms of the model's operations, the module is responsible for two tasks: constructing C from raw urban data sources and implementing data operations $\delta : C \rightarrow C$ that filter, join, and aggregate features while preserving the features as the output units.

Data injection: The spatial database module fetches and parses coastlines, parks, water bodies, streets, and buildings directly from OpenStreetMap. It relies on the Overpass API¹, which supports retrieval of up to 10 million elements filtered by location, object type, tags, proximity, and other criteria. Once fetched, each data layer is stored in an individual DuckDB table with appropriate spatial indexing. After instantiating and initializing the database, to fetch OSM data, the user must define the base search area, the regions to load (e.g., neighborhoods), the output table name, the coordinate format, and the layers to extract. The following example loads all layers of Manhattan Island.

```
const db = new AutkSpatialDb();
await db.init();

await db.loadOsmFromOverpassApi({
  queryArea: {
    search: 'New York',
    load: ['Manhattan Island']
  },
  outputTableName: 'table_osm',
  autoLoadLayers: {
    projection: 'EPSG:3395',
    layers: ['surface', 'parks', 'water', 'roads', 'buildings']
  }
});
```

The module can also load vector data (polylines and polygon sets) provided as GeoJSON, thematic urban data (point sets) in CSV, and raster data (imagery and heatmaps), accommodating both discrete data and continuous spatial fields in GeoTIFF.

Spatial and OLAP queries: Once ingested, the module supports data operations $\delta : C \rightarrow C$ that filter, enrich, and reshape the collection without ever breaking the feature as the unit of representation. Linking urban datasets is essential in visual analytics for hypothesis testing, such as correlating incident reports with nearby street infrastructure. To support this, the spatial database module provides spatial joins (features within

¹<https://overpass-turbo.eu/>

polygons) and nearest neighbor queries (features within a search radius), both with user-defined predicate functions for aggregation.

```
await db.spatialQuery({
  tableRootName: 'neighborhoods',
  tableJoinName: 'noise',
  spatialPredicate: 'JOIN', // 'JOIN', 'NEAREST'
  groupBy: {
    selectColumns: [
      {
        resultField: 'noise',
        column: 'noise_key',
        aggregateFn: 'count' // 'count', 'min', 'max', 'avg'
      }
    ]
  }
});
```

Beyond the high-level query interface, the module incorporates a set of domain-specific optimizations that address both performance and correctness challenges specific to urban datasets (challenges that a generic spatial database would not anticipate). First, the naive approach to nearest-neighbor joins (a brute-force pairwise distance computation across all features) degrades to $O(n \times m)$ on dense urban datasets, making it impractical at metropolitan scale. To avoid this, the implementation pre-filters candidates using a spatial index: before computing exact distances, only features within a bounded expansion of the root geometries are considered, restricting the comparison to a small set of plausible neighbors rather than the full dataset. Second, when the root table contains polygons and the predicate is `NEAREST`, measuring geometry-to-geometry distance is semantically incorrect: adjacent polygons share or nearly share boundaries, causing their distance to collapse to zero. The module detects this case at runtime and switches automatically to centroid-to-centroid distance, which better captures the notion of proximity for analyses between, for instance, neighborhoods and points of interest. This is a default rather than a constraint: the query interface exposes the distance criterion, so the developer can explicitly request centroid-based or full-geometry distance whenever the analysis calls for it.

The buildings layer presents a related correctness challenge as well, this time rooted in how OSM structures its data. In OpenStreetMap, a building is not stored as a sin-

gle polygon but as a set of individual way segments that together define its boundary. Treating these raw segments as independent geometries, as one would for other layers, produces incorrect join results: a point inside a building’s footprint may not be contained by any individual segment, and the same logical building can be counted multiple times. To address this, during ingestion, the individual way segments that compose each building are aggregated into a single composite footprint stored in a dedicated geometry column. The join detects buildings tables and prioritizes this column over the default `geometry`, so that joins operate on the aggregated footprint rather than on the raw way segments.

Queries: In exploratory tasks, the core data dimensions (i.e., spatial location, attribute values or event types, and timestamps) form a triad (where–what–when) where combining any two typically reveals patterns in the third (Ferreira; Poco *et al.*, 2013). For this reason, Autark supports general queries for filtering by location (where), filtering and aggregating by attributes or categories (what), and slicing or grouping across time-varying datasets (when).

Workspaces and persistence: By default, the module operates in a single implicit workspace, so this feature is entirely transparent for most use cases. For scenarios requiring isolation between analysis sessions, Autark organizes its in-browser state through named workspaces, each mapped to a separate DuckDB schema. Switching workspaces with `setWorkspace` makes all subsequent operations (loading, joins, queries) operate exclusively within that schema, so multiple analysis scenarios can coexist in a single database instance without table-name collisions. This addresses a recurring need in urban VA: comparing scenarios (e.g., two cities, two temporal windows) without one polluting the other.

```
await db.setWorkspace('manhattan');
await db.loadOsmFromOverpassApi({ /* ... */ });

await db.setWorkspace('brooklyn');
await db.loadOsmFromOverpassApi({ /* ... */ });
// tables in each workspace are isolated by schema
```

Beyond in-session isolation, the module also make other optimizations, like cache OverpassAPI responses in the browser’s Cache API with a configurable TTL (24 hours by default). This avoids redundant downloads of identical OSM queries across page reloads and partially compensates for the lack of server-side persistence inherent to a backend-free model, where downloads of large urban datasets can otherwise take tens of seconds on

every session.

4.2.2 GPU compute engine

The GPU compute engine also implements the data operation $\delta : C \rightarrow C$ in the feature-centric model. It takes a collection as input, executes analytical or render-based operations in parallel across its features on the GPU, and returns an enriched collection in which each feature carries new computed properties alongside its original attributes. Urban visual analytics workflows typically require this kind of enrichment, combining physical layer features, such as street network segments, with derived thematic values, such as accumulated shadow durations or building visibility scores. The compute engine abstracts low-level WebGPU configuration details, providing a higher-level interface for data binding and shader definition, allowing users to focus on computation logic rather than GPU boilerplate.

Compute programs: The compute engine takes a feature collection C as input. Each feature is mapped to a record in a data buffer consumed by a shader program, which is executed in parallel on the GPU. The metadata associated with each feature is explicitly bound to shader variables. A full execution of the engine produces one (or a list of) output values per feature, which is then written back into the original structure as an additional attribute field, returning an enriched collection that preserves every feature and its original properties while extending each with the computed result. The user supplies the analytical (from simple attribute transformations to complex domain-specific computations) or render-based logic (e.g., pixel counting for feature identification) as a WGSL code fragment that is injected into the shader, while the engine manages compilation, dispatch, and synchronization of GPU workloads. This design supports straightforward shader definition and rapid experimentation with analytical models. The following example illustrates a simple computation that calculates the volume of each building and writes the result as a new feature property.

```
const cp = new AutkComputeEngine();
const enriched = await cp.analytical({
  collection, // input feature collection
  variableMapping: {
    x: 'area',
    y: 'height',
  },
  resultField: 'volume',
  wgslBody: 'return x * y;'
});
```

Heterogeneous attributes and the shader generator: The example above hides a deeper challenge: GPUs operate on fixed-size buffers, but features in urban datasets carry naturally heterogeneous attributes. To handle this, the engine declares three categories of per-feature inputs: scalar values (one `f32` per feature), fixed-length arrays (each feature contributes exactly L elements, packed at offset $\text{idx} \times L$ in a flat buffer), and matrices whose row count may vary across features. This is what enables computations such as shadow casting on building rooftops, where each polygon footprint has a different number of vertices. Beyond per-feature data, the engine also accepts global scalar, array, and matrix *uniforms*, which are uploaded once to the GPU rather than replicated per feature. These are convenient for parameters that are constant across all invocations, such as a sun angle, a day of the year, or model coefficients shared by every feature. The engine distinguishes per-feature accesses (`data[idx]`) from global ones (`data[0u]`) automatically. Multiple computed attributes per execution are also supported: when more than one output column is declared, the shader's return type changes from `f32` to a fixed-size output array and multiple result buffers are allocated in parallel, so correlated quantities (e.g., shadow duration and shadow percentage) can be produced in a single GPU pass. The connecting piece across all of these is automatic shader generation. Given the declarative input mapping, the engine emits the complete WGSL program: binding declarations, type aliases for arrays and matrices, buffer-to-local copy loops, the function signature with its typed parameters, and the `@compute` entry point. The developer writes only the function body. This keeps the developer's mental model aligned with the feature-centric abstraction, since they reason about feature attributes rather than about GPU bindings; the cost is uniformity, as all attributes are coerced to `Float32Array` and non-numeric properties cannot be used directly as inputs without prior encoding.

Working within browser hardware limits: WebGPU exposes conservative defaults

for buffer sizes, which can cause silent failures when input data approaches metropolitan scale. To mitigate this, before creating the GPU device, the engine queries the adapter for its `maxBufferSize` and `maxStorageBufferBindingSize` and explicitly requests the maximum values supported by the underlying hardware. This pushes the practical ceiling for a computation onto the user’s device rather than the runtime defaults. While this is sufficient for prototyping at the scale of cities and neighborhoods, dense metropolitan-scale collections can still hit hardware limits on modest devices (a limitation we revisit in our case studies).

4.2.3 3D map visualization

The 3D map module operationalizes two roles in the feature-centric model. As a renderer, it consumes a collection C and maps each feature’s geometry and attributes to visual encodings on screen. As an interaction surface, it implements the operation $i : C \times E \rightarrow S$: user events trigger a picking operation that identifies the targeted features and emits a selection S of their identifiers, which any other module can consume to drive coordinated updates. Unlike traditional tiled mapping libraries, which treat the map as a collection of pre-rendered layers without providing access to the underlying data, Autark renders all geometry (e.g., coastlines, streets, buildings, and thematic overlays) from the same feature collections that drive computation and interaction, so the feature abstraction is never broken at the rendering boundary. The map module accepts data in GeoJSON and GeoTIFF formats, loaded either from files or from the spatial database module.

```
const collection = ...; // input feature collection

const map = new AutkMap(canvas);
await map.init();

map.loadCollection('neighborhoods', collection);
map.draw();
```

Once loaded, all data (point sets, polylines, polygons, and raster data) are represented as triangle meshes, ensuring a consistent, fast rendering pipeline regardless of the underlying geometry. The viewer renders layers sequentially according to their loading order, enabling a multi-layered composition of heterogeneous data.

Data encoding and map styling: The map visualization module enables mapping feature attributes or aggregated thematic data (e.g. generated through the spatial database

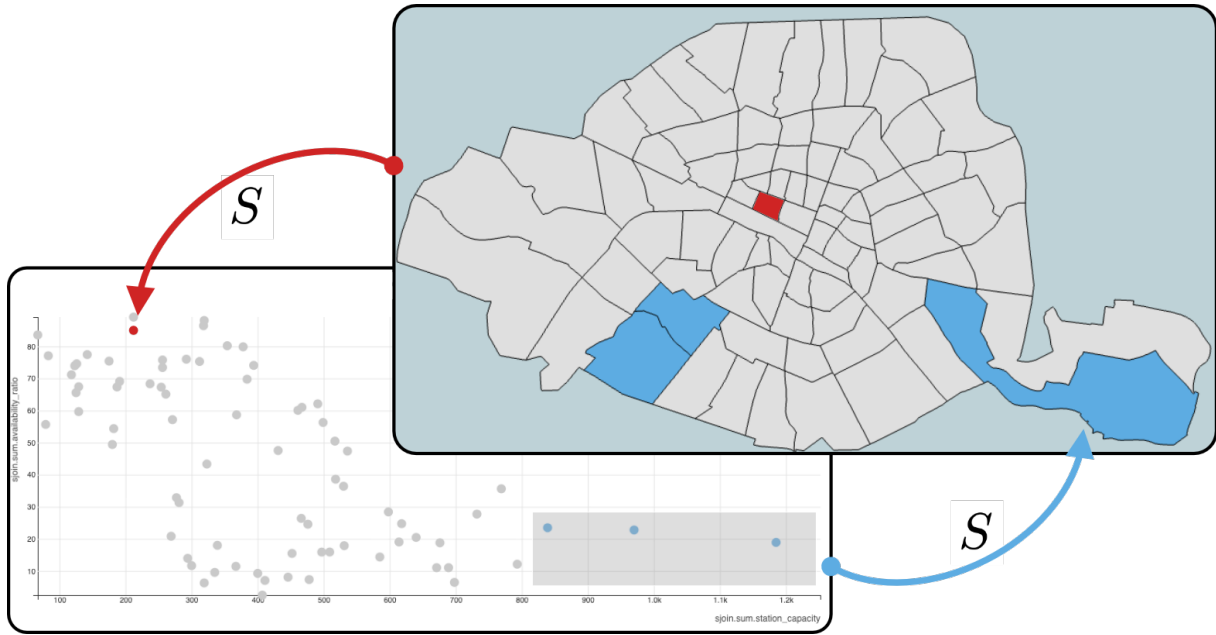


Figure 2: Autark’s map and abstract charts modules facilitate the implementation of linked visualizations. Features selected on the map or brushed in the charts emit events containing selections (S) that other visualizations can react to. In this example, a map of Paris neighborhoods is joined with a city-wide bike station dataset. A scatter plot displaying station capacity versus availability ratio allows users to identify neighborhoods with high capacity but low availability, which are highlighted on the map. Conversely, selecting a neighborhood on the map immediately highlights its corresponding points in the scatter plot.

module), directly to the layer’s color channel. This supports the creation of visualizations such as heatmaps and choropleth maps, and enables visual representation of dynamic highlighting via brushing operations across coordinated views.

```
const collection = ...; // input feature collection

map.updateThematic('neighborhoods', collection, (f: Feature) => {
  return f.properties.sjoin.count.noise;
});
```

The module also allows users to define custom themes by specifying color styles for each layer, supporting both neutral palettes (e.g., for background context) and saturated schemes (e.g., for emphasizing layers of interest).

Feature-Level interactivity: Any layer can be configured to listen to interactions, triggering a picking operation that identifies the selected features by their identifiers. The module then fires an event E carrying the resulting selection S , which other modules can capture through an event-driven architecture. In the example shown in Figure 2,

selecting a neighborhood on the map (Palais Royal, Paris, in red) immediately highlights its corresponding data point in the scatter plot. This bidirectional flow ensures that selections made on the map trigger real-time updates in abstract charts, while interactions within those charts can, in turn, signal the map to highlight specific geographic features.

```
// picking event callback definition
map.events.addListener(MapEvent.PICKING, (selection: number[]) => {
  // event handler
});
```

4.2.4 Abstract charts

The abstract charts module shares the same two roles as the 3D map in the feature-centric model: it consumes a collection C , binding directly to feature attributes for rendering, and implements the interaction operation $i : C \times E \rightarrow S$, translating user events into selections S of feature identifiers that any other module can consume. By operating directly on features and accessing feature attributes from the properties object, the module eliminates the need for data format transformation and keeps a single source for both spatial and abstract visualizations. Autark provides multiple chart types (bar charts, scatterplots, line charts, parallel coordinates, etc.), selected based on the most commonly used chart types in the Urban VA literature.

```
const plot = new AutkChart(plotDiv, {
  type: 'barchart'
  collection, // input feature collection
  labels: { axis: ['name', 'area'], title: 'Neighborhoods area' },
  events: [PlotEvent.CLICK]
});
```

Extensibility: The library enables custom visualizations through a single draw method, within which the developer defines the mapping between feature attributes and visual elements using standard D3.js code and data join patterns. This approach offers two key advantages. First, updates are handled natively. When the underlying collection changes, for instance, after a new spatial join or a GPU computation, the library automatically synchronizes the updated attributes with the existing SVG elements, since it operates on the same feature collection rather than a transformed copy. Second, developers can reuse most of their existing D3.js implementations with minimal integration overhead, as the only requirement is that visual marks store the feature(s) identifier(s) they represent.

User interactions: When instantiating a chart, users can toggle between click and brush interaction modes. In click mode, the selected mark is the element that the user directly clicks. In brush mode, they are the elements in the brushed area. In both cases, the module extracts the selection S and broadcasts it using the same event-driven strategy used by the 3D map module.

```
// click event callback definition  
plot.events.addListener(PlotEvent.CLICK, (selection: number[]) => {  
  // event handler  
});
```

The previous example shows how to register a callback that receives the identifiers of the selected features when a click event occurs. Because S is a set of feature identifiers drawn from the shared collection, it can be applied directly to the map, or to any other module, without translation.

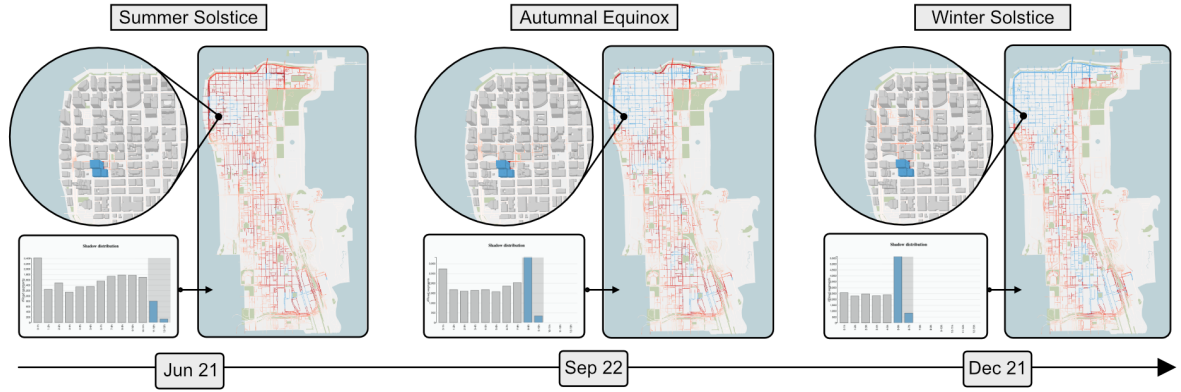


Figure 3: Shadow accumulation in the Chicago Loop during the Summer Solstice (June 21), Autumnal Equinox (September 22), and Winter Solstice (December 21). Developed with Autark the built interactive system integrates linked histograms and a map view with a database of precomputed city-wide shadow data. The histogram shows the distribution of road segments by shadow duration; users can brush specific time ranges on the chart to highlight the corresponding segments in blue on the map. These spatial views show that the number of roads remaining in shadow for extended periods increases as the season progresses toward winter. To provide granular insights, the system can compute shadows cast by a user-selected building onto nearby streets. These local values are calculated in real time using a ray-casting algorithm executed by the GPU compute engine.

5 Evaluation

We evaluate Autark along three dimensions. First, we present a series of usage scenarios demonstrating that Autark can achieve functionality comparable to urban VA systems traditionally built on complex, multi-service infrastructures. Second, we report performance benchmarks measuring data loading and spatial join scalability to assess the viability of a browser-based architecture for urban VA. Third, we conduct a controlled experiment comparing agent-based implementations of urban VA tasks across three conditions (using Autark, using general-purpose libraries, and using UTK ([Moreira; Hosseini; Nipu *et al.*, 2023](#))) to evaluate the toolkit’s impact on agentic development workflows.

5.1 Usage examples

Backend-free remake of Urbane ([Ferreira; Lage *et al.*, 2015](#)):

Figure 4 shows a remake of Urbane built with Autark. Urbane is a multi-resolution (neighborhood and building levels) VA system designed to support data-driven decision-making in urban development, enabling architects and planners to explore neighborhood characteristics across multiple integrated thematic datasets. The original system followed a client-server architecture, relying on preprocessed OSM data for the physical layers and precomputed spatial aggregations to associate thematic data with neighborhoods and buildings. In this reimplementation, all data is dynamically fetched and handled by the system using the spatial database module. OSM data containing the physical layers, a GeoJSON of Manhattan’s neighborhoods, and CSV files of the thematic data of interest are fetched by the toolkit when the system is initialized (Figure 4(A)). Once all data is loaded, we run a render-based program using the GPU compute module to estimate sky exposure, i.e., the percentage of the sky visible along the streets. Then, Autark performs spatial joins between the OSM buildings layer and the neighborhoods layer to determine which buildings belong to each region. Then, thematic and dynamically computed datasets are joined with the neighborhood layer so that each polygon is associated with its

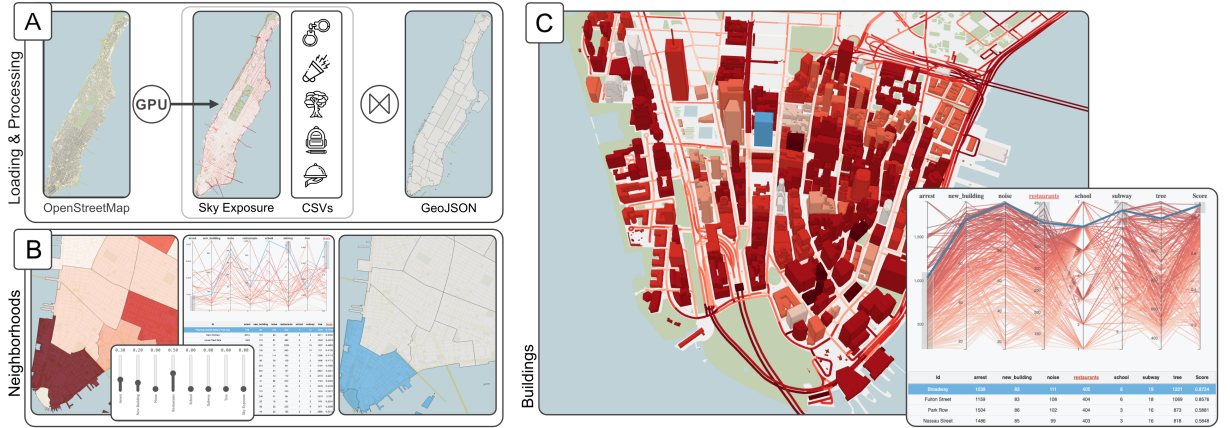


Figure 4: Urbane (Ferreira; Lage *et al.*, 2015) is an urban visual analytics system that supports multi-resolution exploration. It enables users to characterize urban areas based on data and identify sites for new development at the neighborhood and building levels. This figure illustrates a remake of Urbane using Autark. **(A)** On initialization, the system fetches OpenStreetMap data and computes sky exposure values via the GPU. It also loads the Manhattan neighborhood GeoJSON and CSV files containing thematic data, such as arrests and noise complaints, and joins them to the neighborhood polygons. **(B)** Users can define weights for each attribute to compute a score in real-time using the GPU. A choropleth map, a parallel coordinates chart, and a table allow users to explore data distributions and perform interactive multi-attribute filtering. Selecting a neighborhood triggers a drill-down to the building level. **(C)** At the building level, the system joins all datasets to the building footprints within the selected neighborhood. The 3D map displays buildings color-coded by thematic data or scores. Users can identify sites of interest using the same linked and interactive visualizations available at the neighborhood level.

number of arrests, new buildings, trees, noise complaints, restaurants, schools, subway entrances, and the average sky exposure within its area. The loaded data can be initially explored at the neighborhood level using three visualizations (Figure 4(B)): a choropleth map, a parallel coordinates chart, and a table, built using Autark’s 3D-Map and abstract charts modules. All visualizations are linked. Once the user identifies a neighborhood of interest, the system allows them to drill down to the building level and explore the best location for their development. When the drill-down operation is triggered, the system first queries Autark’s database to select buildings within the chosen area, then joins all datasets with the building footprints using a nearest-neighbor spatial operation. The selected buildings and all their associated data can be visualized on the map using the color channel for buildings, as well as in the parallel coordinates and table visualizations previously used at the neighborhood level (Figure 4(C)). When interacting with the charts, the user can highlight the buildings that best fit their selection criteria. Since Autark also allows running code in parallel for each feature of a given layer using the General-purpose GPU computation module, we also implemented in this Urbane remake a dynamic feature score based on user-defined weights through the visual interface (Figure 4(B)). Once the user defines the weights for each attribute of interest, the system instantiates the GPU computation module and recalculates the score for each element in real time. The score computation is available at both the neighborhood and building levels. Although the system remake does not faithfully reproduce the original system, it implements all core functionalities. If compared to the original implementation, which comprised thousands of lines of code, a complex architecture, and a heterogeneous development ecosystem spanning multiple programming languages and dependencies, the Autark-based implementation achieves the same functionality in under 350 lines of code, runs entirely in the browser, relies solely on TypeScript, and is straightforward to deploy and extend.

Accumulated shadows in Chicago: Accumulated gross shadow is a measure that quantifies the total time a specific location is in shade during a given time interval (Miranda; Doraiswamy *et al.*, 2019). By capturing the cumulative duration of shadows across regions, this metric helps planners evaluate the impact of building developments on the environmental quality of public spaces, informing decisions about building height regulations and the preservation of sunlight access. Using Autark, we developed an application to analyze shadow patterns in the Chicago Loop during three seasonal markers (Figure 3): the summer solstice (June 21), the autumnal equinox (September 22), and the winter solstice (December 21). The workflow began by loading OpenStreetMap data layers for roads

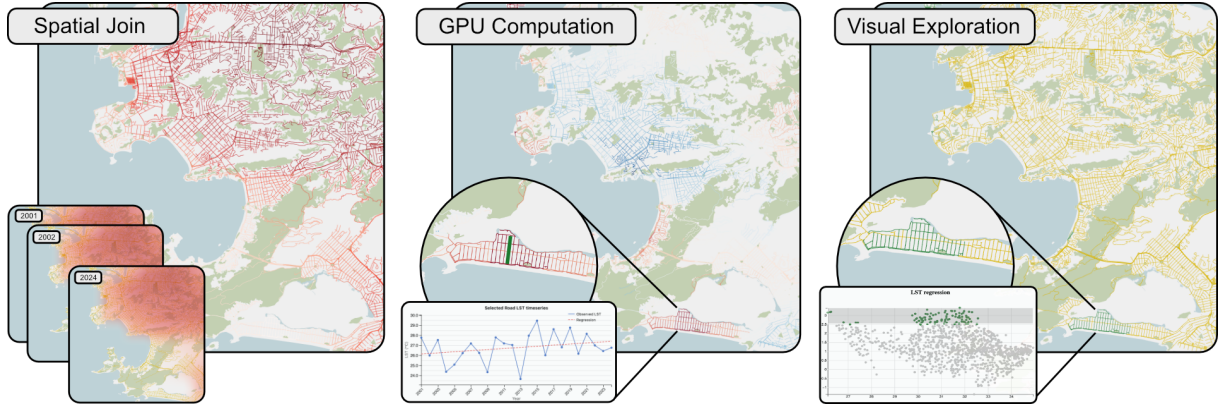


Figure 5: Analysis of urban heat trends in Niterói (2001–2024) using Autark. The system processes NASA LST raster data to enrich road segments with average summer temperature over the years (left column). Autark’s compute engine, in parallel, calculates linear regressions for each road segment to map temperature trend (slope) and base heat (intercept) across the city. Line charts showing the heat data and the regression lines for picked roads can be produced dynamically (middle column). Finally, a scatter plot showing the slope and intercept values of all roads in the city is constructed to enable global exploration of heat areas. The chart’s brushing interaction allows users to highlight areas of significant warming on the map, such as the Piratininga neighborhood, where recent urban expansion correlates with rising temperatures (right column).

and buildings, along with the accumulated shadows dataset¹, which encodes the total daily duration of solar occlusion for each street segment across the three dates of interest. We then leveraged Autark’s spatial database module to join the shadow values with the street network data, associating each road segment with its corresponding shadow duration. Finally, we used the Autark abstract charts module to generate a histogram of the number of street segments in shadow for durations ranging from 1 to several hours per day, facilitating a seasonal comparison of urban solar access. The system utilizes Autark’s support for linked visualizations to implement a brush interaction on the histogram. As illustrated in Figure 3, an analyst can select specific time ranges to immediately identify and highlight corresponding road segments in blue on the map, enabling direct spatial inspection of the selected distribution. These coordinated views reveal a clear seasonal trend: the number of roads experiencing prolonged solar occlusion increases significantly as the year progresses toward the winter solstice. To take the analysis further, the application leverages the 3D map picking interaction and the Autark compute module to allow users to select a specific building and calculate its individual contribution to the accumulated shadows on surrounding streets. This feature, shown in the circular zoom insets in Figure 3, employs a ray-casting approach at an *hourly resolution* for the day of interest. More precisely, for each 20-meter road segment, the system casts a ray toward

¹<https://osf.io/4tqp9/overview>

the sun at every hour; if the ray intersects the selected building, it is recorded as a potential shadow source for that segment at that time. These local values are calculated in real time, allowing analysts to interactively probe the shadow contribution of individual structures. We observe that this computation result represents a lower bound, as other structures could be positioned between the building of interest and the segment. We argue that this approach ensures the estimated contribution is never lower than reality.

Heat islands in Niterói: This use case demonstrates Autark’s ability to process raster geospatial datasets. To do so, we developed an application to monitor trends in urban land temperatures. The analysis uses NASA Land Surface Temperature (LST) data covering the period from 2001 to 2024 for the city of Niterói². We preprocessed the original raster files to create a multi-band TIFF, with each band storing the average summer temperature (i.e., data from December, January, and February) for a specific year, resulting in a compact representation of the dataset’s full temporal extent. As illustrated in the first column of Figure 5, we first use the spatial database module of Autark to load the multi-band TIFF file and perform a spatial join between the road network and the temperature time series, enriching each street segment with a time series of summer heat values that captures its thermal history over the observation period. Next, to identify areas with increasing surface temperatures, we used Autark’s compute engine to calculate linear regressions of surface temperature values for each road segment in parallel. The regression is computed independently for each segment, taking full advantage of the GPU’s parallelism to process the entire road network simultaneously. The slope of the regression indicates the trend in temperature variation, while the intercept encodes the base temperature values and may help locate the hottest areas. In the middle column of Figure 5, we see the road network of Niterói colored by the regression slope (using a diverging scale, from blue to red). Negative slope values mean that temperatures in the region decreased from 2001 to 2024. Positive values mean that the temperature is increasing in the region. We also used the feature-picking functionality of Autark’s 3D-map module to implement a dynamically updated line chart that displays the temperature time series and the fitted regression line for segments of interest, enabling detailed investigation of localized thermal trends and specific anomalous situations. Finally, we created a scatterplot to facilitate a global exploration of heat areas over the city. The chart implements a brush interaction that allows users to highlight roads on the map based on the slope and intercept values of interest. By jointly analyzing the slope and intercept of each road segment, users can distinguish between areas that are both hot and warming from those that are warming

²<https://shorturl.at/ROG5v>

but starting from a lower baseline. Interactively exploring the visualizations, we identified that one of the regions with the highest increases in temperature was the Piratininga neighborhood (see the last column of Figure 5), where intense urban expansion and the construction of several new buildings have occurred in recent years.

5.2 Performance

Three performance experiments were conducted on an M1 Mac Studio (32GB RAM) to evaluate the feasibility of Autark’s fully client-side approach. The first assessed loading time by incrementally loading Manhattan neighborhood data from 1 to 10 areas (2,000–20,000 features). The second evaluated spatial join performance using 28 neighborhood polygons joined against approximately 4.5 million georeferenced 311 service records, sampled at 10% increments³. The third evaluated GPU compute performance across approximately 11,000 segments of Manhattan’s road network. Each test was run 10 times to assess robustness.

For loading time, at 20,000 features, the total time averages 6.0s (sd=460ms), with database ingestion accounting for 4.9s (sd=295ms) and 3D map data loading for 1.1s (sd=180ms), yielding a cost of $\approx 0.3\text{ms}$ per feature. For spatial joins, processing the full 4.5 million records requires, on average, 2.6s (sd=6ms), corresponding to $\approx 0.58\mu\text{s}$ per record. Both metrics scale linearly with dataset size, suggesting predictable behavior for larger inputs. Render-based computation test used a $6,784 \times 6,784$ target to store 11,224 tiles (64×64 pixels), consuming 185 MB of GPU memory and ran on average in 4.2s (sd=45ms).

These results are particularly noteworthy given that Autark runs entirely in the browser, where such operations would traditionally require a backend, and that no optimizations such as data streaming were applied. Given that modern browsers can allocate several gigabytes of memory per tab, these results suggest that fully client-side architectures are a feasible alternative for urban visual analytics systems.

5.2.1 Agentic urban VA development

The development of Urban VA systems is undergoing a paradigm shift in which code is increasingly written by agents, yet the resulting codebase must remain understandable and maintainable by humans who define requirements and verify correctness. This raises

³<https://shorturl.at/HylCQ>

a key question: how does a toolkit’s design influence the quality of agent-generated code? To investigate this, we evaluated Autark’s effectiveness through a controlled experiment comparing agent-based implementations of well-scoped urban VA systems across three conditions: one constrained to Autark’s modules; a baseline that allowed the agent to use any general-purpose libraries available online (except Autark); and one constrained to UTK (Moreira; Hosseini; Nipu *et al.*, 2023), the most closely related urban VA toolkit, which sits at the opposite end of the design space with its declarative grammar and Python preprocessing backend (Section 2.3). We analyze the results along two complementary axes. The first is the *structural quality* of the generated code (complexity, maintainability, and readability), which we report only for the Autark and baseline conditions: both emit imperative TypeScript and are therefore measured on the same ruler. The second is the *agentic effort* required to reach a working application (iterations, human intervention, wall-clock time, and monetary cost), which we report for all three conditions, and where UTK becomes directly comparable. As we discuss below, this split is itself a finding: Autark wins decisively on the first axis, while the second axis separates both imperative approaches from UTK’s declarative grammar plus backend.

Experimental setup: We designed five urban VA tasks of increasing complexity, each requiring a combination of data ingestion, spatial operations, and interactive visualization. The first task, the simplest, requires loading multiple OpenStreetMap Layers and coloring specific layers based on a spatially joined attribute from an external thematic dataset. Each successive task adds requirements: additional thematic datasets, multiple spatial joins, linked charts, and GPU-accelerated computations. All the tasks, prompts, and experiment results are available in the supplemental material of this paper on the GitHub⁴. Each task was implemented by an AI coding agent (Claude Code with the Opus 4.8 model); an earlier run of the Autark and baseline conditions on the previous-generation Opus 4.6 yields the same qualitative result. For all conditions, we prompted the task definition and used the same model configuration. Each trial was conducted independently, with no conversation history carried over between tasks. To ensure a fair comparison, all conditions shared the same stop criteria: after generating the project, the agent was required to validate that the system fully compiles, builds, and serves without errors. For the Autark condition, the agent was provided with Autark’s documentation as its primary context and instructed to use only Autark’s API. For the baseline condition, the agent was explicitly told not to use Autark to build the system. For the UTK condition, the agent was given UTK’s official documentation and grammar reference as its

⁴<https://github.com/urban-toolkit/autark-experiments/tree/utk-test>

primary context, instructed to use only UTK, and additionally had to author the Python preprocessing workflow that ingests and persists data into UTK’s `.utk` files before the grammar can be rendered.

Functional issues: Our first goal in analyzing the experiments was to assess whether the generated systems behave as specified. For each task and condition, we manually inspected the output application against its functional requirements, such as layer rendering, data encoding, spatial operations, and view coordination. The first issue concerns layer representation. Without Autark, the agent frequently combines multiple mapping libraries within the same application to render different urban layers. As shown in the baseline column of Figure 6, this results in a fragmented visual stack, with layers from different rendering libraries coexisting on the same canvas. Beyond the visual inconsistency, it introduces a functional limitation: picking operations cannot traverse libraries, making it impossible to select or interact with features on layers that are occluded by elements rendered by a different backend. Autark avoids this entirely by routing all layers through a single rendering pipeline, ensuring that picking and interaction work uniformly across the full layer stack. The second issue concerns layer processing. Autark’s spatial database module processes raw OpenStreetMap data through a unified pipeline that reconstructs semantically coherent geometries, assembling building footprints from their constituent way segments before storing them as individual features. Examples from the baseline condition lack this consolidation step. As a result, parts of the same physical structure are treated as independent geometries, as illustrated in the baseline column of Figure 6. This inconsistency hinders geometry-dependent computations, as thematic attributes might be assigned to parts of the layer elements, and physical properties, such as building areas, become unreliable. These processing issues also affect rendering quality. As shown in Figure 6, the baseline condition produces visible mesh artifacts, such as fragmented polygons and overlapping edges that result from operating on raw, unconsolidated geometry. Autark’s unified processing pipeline reconstructs coherent geometries before rendering, ensuring that the visual output is free of such artifacts. The UTK condition tells a different and more nuanced story. Because UTK is a mature, purpose-built urban toolkit, its consolidation and rendering of building geometry ultimately matched Autark’s quality, without the fragmentation seen in the baseline (Figure 6). The difference is in the path to that result rather than the result itself: as detailed below, UTK required error-fix iterations and a manual intervention to converge, whereas Autark produced a correct application in a single interaction. The points that follow examine this gap in more detail, quantifying the effort each condition demanded and the friction behind UTK’s iterations.

Code generation metrics: Beyond functional correctness, we also profile the generated code to characterize its structural properties. Even in an agentic workflow, developers must be able to read, navigate, and understand the generated codebase in order to verify the agent’s design decisions. To this end, we computed multiple code metrics that serve as proxies potentially indicative of code understandability and ease of maintenance. Per-trial results are available in the supplemental material. Autark-based implementations average 222 lines of code (LOC) compared to 787 for the baseline condition ($\approx 3.5\times$) and are consistently contained in a single source file, whereas the baseline spreads across 7.2 files on average. Each external dependency adds API surface that the developer must understand, version constraints that must be managed, and potential points of failure that complicate debugging, all of which compound the maintenance burden of the generated codebase; here the baseline introduces more dependencies than Autark (3.6 vs. 2.4 on average, as Autark-based code mainly imports its own modules), though the gap is modest. Cyclomatic complexity measures the number of independent paths through a program’s control flow, serving as a widely used proxy for how difficult code is to understand and test (McCabe, 1976). Higher values indicate more branching logic that developers must trace when verifying correctness. Autark code averages a cyclomatic complexity of 27 compared to 114 for the baseline ($\approx 4.2\times$), indicating that the toolkit’s domain-specific abstractions absorb the branching logic that the agent must otherwise generate from low-level primitives. Taken together, these metrics suggest that Autark’s domain-specific abstractions not only reduce the volume of generated code but also produce codebases that are structurally simpler and more uniform. As demonstrated in the usage scenarios (Section 5.1), these gains do not come at the cost of expressiveness, as Autark achieves feature-parity with traditional urban VA systems while producing code that is more compact. While these proxies do not guarantee maintainability, they consistently lower the barrier for developers who need to inspect, verify, and extend the agent-generated codebase. We deliberately restrict these structural metrics to the Autark and baseline conditions, and exclude UTK. The metrics above are proxies for the readability and maintainability of imperative application code, and are only meaningful when comparing artifacts of the same kind: Autark and the baseline both produce imperative TypeScript, which makes for a fairer comparison. UTK’s deliverable is categorically different. A UTK application is primarily a declarative JSON grammar plus a Python preprocessing step, with almost no hand-written control flow to measure. The cyclomatic complexity of a JSON specification is near-zero by construction, a line count measures a data format rather than logic, and the dependencies are fixed by the UTK runtime rather than chosen by the agent. Repor-

ting these side-by-side would flatter UTK for reasons unrelated to the quality of what the agent produced. This is precisely the declarative-versus-imperative trade-off discussed in Section 2.3: declarative grammars are concise and reproducible by design, which is their strength, and exactly why a line (or complexity) count is the wrong lens for them. The appropriate lens for UTK is how much effort it took the agent, and the human, to reach a working result, which is the axis we turn to next, and where all three conditions are comparable.

Cost and effort of agentic development: The structural metrics compare the *artifact*; this final axis compares the *process* of reaching it, and is the one on which all three conditions can be measured. We quantify effort along three dimensions: the number of iterations to a working build, whether human intervention beyond reporting errors was required, and the time and monetary cost the agent consumed. We define an *iteration* precisely as a round in which the agent explicitly ended its run claiming success, the output was then manually tested, an error surfaced, and that error was fed back into the same session to let the agent continue. By this definition, the Autark and baseline conditions were one-shot in every task (the agent declared completion, the application was tested, and it worked), whereas UTK required more than one iteration in most tasks, averaging 2.6 rounds. The iteration count is thus itself a discriminating metric, not merely a driver of cost. One detail is worth an aside here, since it shaped UTK’s effort. Every UTK trial hit a consistent data-fetching bug that drove the iteration count up sharply, and the agent struggled to identify its cause: the Python black-box behind UTK’s preprocessing made it hard to understand what was going wrong. Once we diagnosed the problem, and so as not to be unfair to UTK, we added targeted instructions to the prompt to fix this specific error, which let us better assess the iterations needed to reach the final output.

We derive the monetary cost directly from the agent’s own execution log rather than estimating it. Claude Code emits a structured log whose final result event reports the complete token accounting for the run, fresh input tokens, output tokens, and cache reads and writes. Multiplying each token category by the official Opus 4.8 per-token prices in effect at the time of execution yields a single USD figure that summarizes the total computational effort the agent expended to produce the application.⁵ Because error-fix iterations accumulate within the same session, each additional round contributes its own tokens and wall-clock time to these totals; the cost and time metrics therefore already encode the iteration burden, and a condition that needs more rounds is automatically

⁵Opus 4.8 pricing at execution time: \$5/MTok input, \$25/MTok output, \$0.50/MTok cache read, and \$6.25–\$10/MTok cache write. Costs are model- and date-specific.

Table 1: Agentic effort to reach a working build, averaged over the five tasks. Time and cost are derived from the agent’s execution log (token usage $\times$ Opus 4.8 pricing at execution time); error-fix iterations roll up into these totals. UTK’s mean is inflated by a costly outlier (app2, a 68.7-minute, \$24.82 run), but even its median cost (\$6.52) remains well above the imperative conditions ($\sim 2.7\times$).

Condition	Mean time (s)	Mean cost (USD)	Iterations to working build	Human fix
Autark	416	2.45	1 (one-shot, all tasks)	no
Baseline	426	2.17	1 (one-shot, all tasks)	no
UTK	1800	10.06	2.6 (range 1–4)	yes

reflected as a higher cost. Table 1 reports the per-condition means alongside the iteration and intervention story.

The cost axis does *not* separate Autark from the unconstrained baseline: the two are essentially tied (~ 400 – 430 s and $\sim \$2.2$ – 2.5 per application), with the baseline marginally cheaper on average. We therefore do not claim a cost advantage for Autark over general-purpose coding. What the axis separates is UTK from both imperative approaches, along two reinforcing lines: the *absolute cost* and the *predictability and human cost*. On average UTK is $\sim 4.3\times$ slower and ~ 4.1 – $4.6\times$ more expensive. Part of this mean is inflated by a costly outlier (a \$24.82 run), but the gap does not vanish once outliers are set aside: UTK’s median run still costs \$6.52, roughly $2.7\times$ the imperative conditions, and only on the single simplest task did it come in cheaper than Autark. On top of this systematically higher cost, its effort blows up further when it struggles. The more decisive difference, however, remains qualitative. Both imperative conditions were one-shot across all five tasks and never required a human to step in beyond reporting errors, whereas four of the five UTK tasks forced the agent to iterate and at least one required manual intervention. A toolkit whose iteration count is one and whose agentic cost is bounded is far easier to plan around than one that is competitive on average but unbounded on the hard cases. Taken together, the two axes are the payoff of this evaluation: on code structure Autark is far simpler than the baseline (and UTK is not comparable), while on agentic effort both imperative approaches comfortably beat UTK’s declarative grammar plus backend. Autark thus obtains code as structurally simple as a purpose-built toolkit ought to produce, at an agentic cost indistinguishable from unconstrained general-purpose coding and well below UTK. This supports the thesis of Section 2.3: a clear programmatic API over standard representations is a cheaper natural-language target than a bespoke grammar coupled to a proprietary preprocessing pipeline.

Why imperative libraries are an easier target: We can only hypothesize about the

mechanism behind the gap between UTK and the imperative conditions, but one explanation is consistent with what we observed. A code-based library keeps documentation, source, and generated artifact in the *same modality*: all three are TypeScript. When the documentation does not cover a given task, the agent can degrade gracefully by reading the library’s own source, which maps almost one-to-one onto the code it must write. A specification-based toolkit breaks this homogeneity. The artifact the agent produces is a declarative grammar, but the ground truth for its behavior lives in an interpreter written in another language, UTK’s Python preprocessing backend, so the same fallback, reading the source, crosses a modality boundary and yields far less. The data-fetching bug discussed above is concrete evidence of this: the failure originated in the opaque Python step, and the agent struggled precisely because it could not bridge from the grammar it was writing to the backend that was failing. The effect is not specific to Autark: the baseline is also code-based and was likewise one-shot, which is what we would expect if modality homogeneity, rather than any property unique to Autark, is what separates the imperative conditions from UTK.

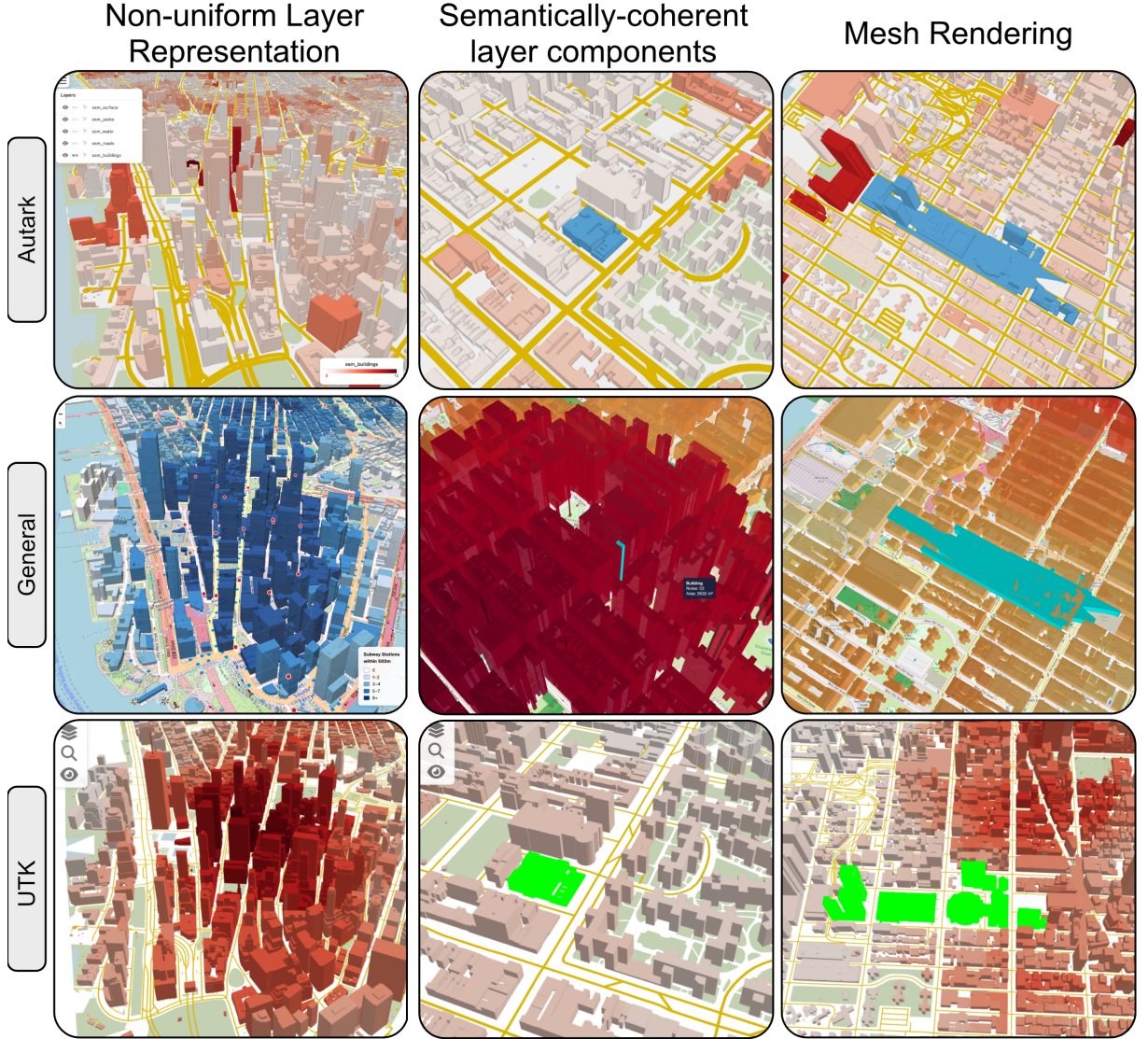


Figure 6: Comparison of agent-generated outputs across the three conditions: Autark, the baseline (general-purpose libraries), and UTK. The baseline exhibits the functional issues discussed in the text, combining multiple rendering libraries into a fragmented visual stack, treating parts of the same physical structure as independent geometries, and producing mesh artifacts such as fragmented polygons. Autark reconstructs semantically coherent building footprints and renders them artifact-free in a single interaction. UTK ultimately reaches comparable output quality, but only after several error-fix iterations and a manual backend fix.

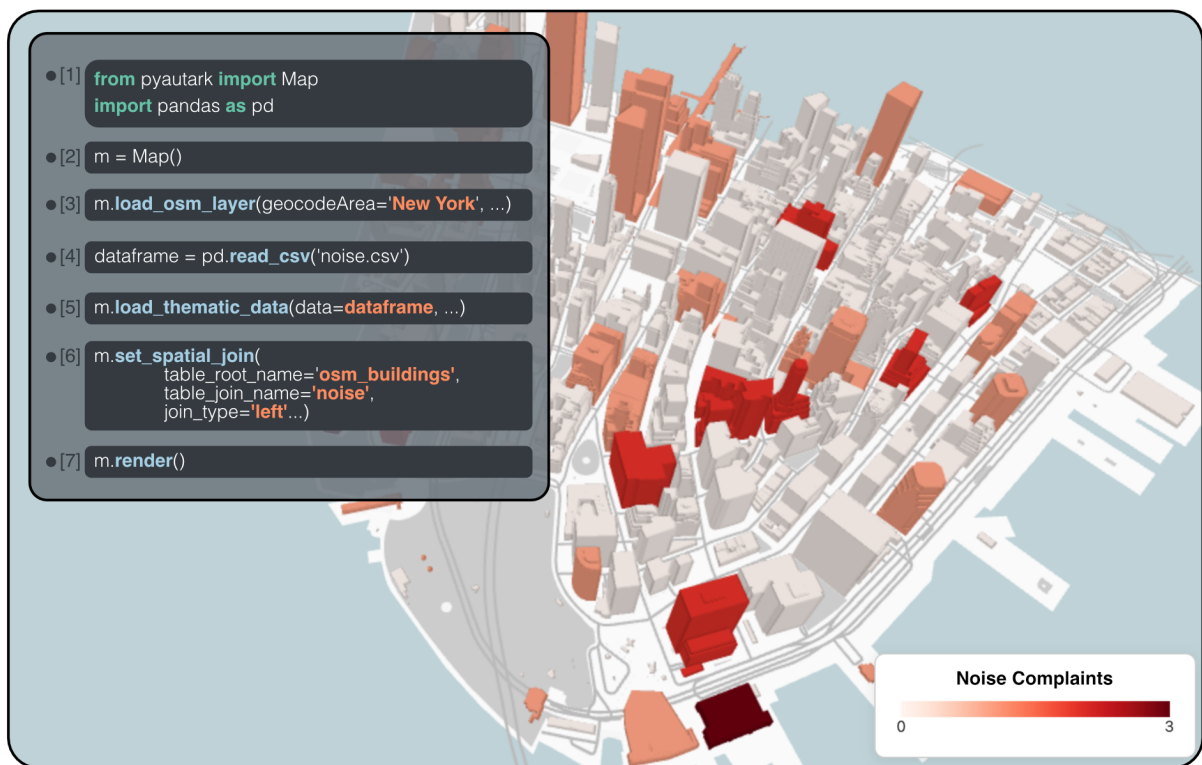


Figure 7: Autark wrapper for Jupyter notebooks, illustrating how its self-contained, client-side architecture facilitates integration with diverse computational environments and community standards.

6 Conclusion

In this dissertation we presented Autark, a fully client-side toolkit for rapid prototyping of urban visual analytics systems that consolidates spatial data management, GPU-accelerated computation, and multi-scale visualization into a single, in-browser codebase organized around a feature-centric model. In contrast to the client-server pipelines that characterize most urban VA systems, Autark ingests, queries, and computes entirely at runtime in the browser, making the resulting applications dynamic, trivial to deploy, and inherently shareable. We evaluated the toolkit along three complementary dimensions. A series of usage scenarios showed that systems traditionally built on complex, multi-service infrastructures, such as a remake of Urbane, accumulated-shadow analysis in Chicago, and urban heat-island monitoring in Niterói, can be reproduced with feature parity in a fraction of the code and with no backend. Performance benchmarks showed that data loading, spatial joins, and GPU-based computation scale linearly and remain interactive within a single browser tab, supporting the viability of a fully client-side architecture for urban VA. Finally, a controlled experiment on agentic development compared Autark against an unconstrained baseline and UTK (Moreira; Hosseini; Nipu *et al.*, 2023), finding that Autark produces structurally simpler and accurate code than the baseline while reaching a working application with far less effort and more predictability than UTK.

This last dimension supports a broader thesis developed throughout the dissertation: that a clear, well-scoped programmatic API over standard representations is a more effective natural-language target for code generation than a declarative grammar coupled to a preprocessing backend. The evaluation separates this claim along two axes: structural quality, on which Autark clearly improves over the baseline, and agentic effort, on which both imperative approaches converge in a single interaction and at a bounded cost, whereas UTK, at the opposite end of the design space (Section 2.3), needed repeated error-fix iterations and manual intervention. A plausible explanation is modality homogeneity: a code-based library keeps documentation, source, and generated artifact in the same modality, so an agent can degrade gracefully by reading the library’s own source

when documentation runs out, whereas a specification-based toolkit forces the agent to bridge from the grammar it writes to an interpreter living in another language. As natural language becomes a primary interface for assembling VA systems, this property reframes a trade-off that, when UTK was designed, justifiably favored declarative grammars for accessibility.

Community adoption: Autark’s self-contained, backend-free architecture facilitates integration with diverse computational ecosystems without requiring specific development environments or deployment stacks. Figure 7 shows Autark used within a Jupyter notebook via a Python wrapper (`pyautark`¹), analyzing buildings affected by noise complaints in New York’s Financial District. Beyond Jupyter, Autark is compatible with cloud-hosted platforms such as Google Colab, reactive notebook environments such as Observable (Bostock; Conlen *et al.*, 2024), and WebXR platforms for immersive analytics (Wagner *et al.*, 2024).

Limitations: Autark’s fully client-side architecture, while effective for rapid prototyping, introduces inherent scalability constraints. The toolkit is bound by the memory and compute resources available in a single browser tab. For the datasets explored in this paper, this proved sufficient, with linear scaling behavior and interactive response times (Section 5.2). However, for terabyte-scale datasets, a browser-based approach is not a viable alternative to server-backed architectures with dedicated database infrastructure. Additionally, the set of abstract chart types currently provided by Autark was informed by a decade of experience building urban VA systems and by a thorough review of the urban VA literature, but it is not exhaustive. Autark partially addresses this by providing an extensibility mechanism that allows developers to define custom visualization code while retaining integration with the feature-centric model. A more comprehensive built-in chart library remains a direction for future work. Our empirical findings should also be read against the scope of the study. The controlled experiment rests on five tasks implemented by a single coding agent and model; while an earlier run on the previous-generation model yields the same qualitative result, broader task suites, additional agents, and human-in-the-loop workflows remain to be tested. Finally, the structural code metrics are meaningful only for the two imperative conditions, and we make no claim of a cost advantage over unconstrained general-purpose coding: Autark’s gains are structural, and on the agentic-effort axis they lie in predictability rather than in a lower absolute mean cost.

¹<https://github.com/urban-toolkit/pyautark>

Future work: We envision two main directions for future work. First, we plan to leverage Autark to better understand the construction of VA systems within agentic ecosystems. The evaluation presented in this paper adopted a purely agentic approach, where the agent was solely responsible for generating the entire codebase. In practice, however, the development process is more nuanced: developers may co-write code alongside agents, selectively delegate specific components, or write entire modules themselves. We plan to conduct a large-scale user study using Autark to investigate how developers interact with agents across this spectrum of collaboration, examining how toolkit design influences not only the quality of the output but also the dynamics of the human-agent development process. Second, we plan to explore integrating Autark into immersive environments. Autark’s web-first, client-side architecture is inherently compatible with WebXR, and its feature-centric model provides a natural foundation for immersive urban analytics. This direction has the potential to reduce barriers to developing immersive VA applications, which currently require substantial additional engineering effort beyond an already complex urban VA stack.

REFERENCES

AI, Kuangshi; TANG, Kaiyuan; WANG, Chaoli. NLI4VolVis: Natural Language Interaction for Volume Visualization via LLM Multi-agents and Editable 3D Gaussian Splatting. **IEEE Trans. Vis. Comput. Graph.**, 2025. DOI: [10.1109/TVCG.2024.3520556](https://doi.org/10.1109/TVCG.2024.3520556).

ANDRIENKO, Gennady *et al.* Visual Analytics of Mobility and Transportation: State of the Art and Further Research Directions. **IEEE Trans. Intell. Transp. Syst.**, v. 18, n. 8, p. 2232–2249, 2017. DOI: [10.1109/TITS.2017.2701893](https://doi.org/10.1109/TITS.2017.2701893).

BONADIA, Sandro *et al.* Visual Analytics Using Heterogeneous Urban Data. *In: PROC. SIBGRAPI. [S. l.: s. n.]*, 2023. p. 1–6. DOI: [10.1109/SIBGRAPI59091.2023.10347156](https://doi.org/10.1109/SIBGRAPI59091.2023.10347156).

BOOTH, Bob; MITCHELL, Andy *et al.* **Getting Started with ArcGIS**. [S. l.]: Esri, 2001.

BOSTOCK, Michael; HEER, Jeffrey. ProtoVis: A Graphical Toolkit for Visualization. **IEEE Trans. Vis. Comput. Graph.**, v. 15, n. 6, p. 1121–1128, 2009. DOI: [10.1109/TVCG.2009.174](https://doi.org/10.1109/TVCG.2009.174).

BOSTOCK, Michael; OGIEVETSKY, Vadim; HEER, Jeffrey. D3: Data-driven Documents. **IEEE Trans. Vis. Comput. Graph.**, v. 17, n. 12, p. 2301–2309, 2011. DOI: [10.1109/TVCG.2011.185](https://doi.org/10.1109/TVCG.2011.185).

BOSTOCK, Mike; CONLEN, Matthew *et al.* **Observable: The Collaborative Data Canvas**. [S. l.: s. n.], 2024. <https://observablehq.com>. Accessed: 2026-03-21.

CHEN, Shaoyu *et al.* UrbanRama: Navigating Cities in Virtual Reality. **IEEE Trans. Vis. Comput. Graph.**, v. 28, n. 12, p. 4685–4699, 2021. DOI: [10.1109/TVCG.2021.3099012](https://doi.org/10.1109/TVCG.2021.3099012).

DECK.GL. [S. l.: s. n.], 2024. <https://deck.gl/>.

DENG, Zikun *et al.* AirVis: Visual Analytics of Air Pollution Propagation. **IEEE Trans. Vis. Comput. Graph.**, v. 26, n. 1, p. 800–810, 2019. DOI: [10.1109/TVCG.2019.2934670](https://doi.org/10.1109/TVCG.2019.2934670).

- DIBIA, Victor. LIDA: A Tool for Automatic Generation of Grammar-agnostic Visualizations and Infographics Using Large Language Models. *In: PROC. ACL. [S. l.: s. n.]*, 2023. p. 113–126. DOI: [10.18653/v1/2023.acl-demo.11](https://doi.org/10.18653/v1/2023.acl-demo.11).
- DORAISWAMY, Harish *et al.* Spatio-temporal Urban Data Analysis: A Visual Analytics Perspective. **IEEE Comput. Graph. Appl.**, v. 38, n. 5, p. 26–35, 2018. DOI: [10.1109/MCG.2018.2852702](https://doi.org/10.1109/MCG.2018.2852702).
- ELMQVIST, Niklas; TSIGAS, Philippos. A Taxonomy of 3D Occlusion Management for Visualization. **IEEE Trans. Vis. Comput. Graph.**, v. 14, n. 5, p. 1095–1109, 2008. DOI: [10.1109/TVCG.2008.59](https://doi.org/10.1109/TVCG.2008.59).
- FERREIRA, Leonardo; MOREIRA, Gustavo; HOSSEINI, Maryam *et al.* Assessing the Landscape of Toolkits, Frameworks, and Authoring Tools for Urban Visual Analytics Systems. **Comput. Graph.**, v. 123, p. 104013, 2024. DOI: [10.1016/j.cag.2024.104013](https://doi.org/10.1016/j.cag.2024.104013).
- FERREIRA, Leonardo; MOREIRA, Gustavo; MIRANDA, Fabio. VA-Blueprint: Uncovering Building Blocks for Visual Analytics System Design. **IEEE Trans. Vis. Comput. Graph.**, 2025. DOI: [10.1109/TVCG.2024.3456345](https://doi.org/10.1109/TVCG.2024.3456345).
- FERREIRA, Nivan; LAGE, Marcos *et al.* Urbane: A 3D Framework to Support Data Driven Decision Making in Urban Development. *In: PROC. VAST. [S. l.: s. n.]*, 2015. p. 97–104. DOI: [10.1109/VAST.2015.7347636](https://doi.org/10.1109/VAST.2015.7347636).
- FERREIRA, Nivan; POCO, Jorge *et al.* Visual Exploration of Big Spatio-temporal Urban Data: A Study of New York City Taxi Trips. **IEEE Trans. Vis. Comput. Graph.**, v. 19, n. 12, p. 2149–2158, 2013. DOI: [10.1109/TVCG.2013.226](https://doi.org/10.1109/TVCG.2013.226).
- GAO, Tong *et al.* DataTone: Managing Ambiguity in Natural Language Interfaces for Data Visualization. *In: PROC. UIST. [S. l.: s. n.]*, 2015. p. 489–500. DOI: [10.1145/2807442.2807478](https://doi.org/10.1145/2807442.2807478).
- GARCÍA-ZANABRIA, Germain *et al.* CriPAV: Street-Level Crime Patterns Analysis and Visualization. **IEEE Trans. Vis. Comput. Graph.**, v. 28, n. 12, p. 4000–4015, 2021. DOI: [10.1109/TVCG.2021.3111146](https://doi.org/10.1109/TVCG.2021.3111146).
- GUO, Hanqi *et al.* TripVista: Triple Perspective Visual Trajectory Analytics and its Application on Microscopic Traffic Data at a Road Intersection. *In: PROC. PacificVis. [S. l.: s. n.]*, 2011. p. 163–170. DOI: [10.1109/PACIFICVIS.2011.5742387](https://doi.org/10.1109/PACIFICVIS.2011.5742387).
- HEER, Jeffrey; CARD, Stuart K.; LANDAY, James A. Prefuse: A Toolkit for Interactive Information Visualization. *In: PROC. CHI. [S. l.: s. n.]*, 2005. p. 421–430. DOI: [10.1145/1054972.1055031](https://doi.org/10.1145/1054972.1055031).

- HEER, Jeffrey; MORITZ, Dominik; PECHUK, Ron. Mosaic: An Architecture for Linking Databases and Scalable Interactive Visualizations. *In: PROC. SIGMOD. [S. l.: s. n.]*, 2025. p. 123–126. DOI: [10.1145/3632410.3632420](https://doi.org/10.1145/3632410.3632420).
- LINS, Lauro; KLOSOWSKI, James T.; SCHEIDEGGER, Carlos. Nanocubes for Real-time Exploration of Spatiotemporal Datasets. **IEEE Trans. Vis. Comput. Graph.**, v. 19, n. 12, p. 2456–2465, 2013. DOI: [10.1109/TVCG.2013.179](https://doi.org/10.1109/TVCG.2013.179).
- LIU, Zhicheng; JIANG, Biye; HEER, Jeffrey. imMens: Real-time Visual Querying of Big Data. *In: COMPUT. Graph. Forum. [S. l.: s. n.]*, 2013. v. 32, p. 421–430. DOI: [10.1111/cgf.12129](https://doi.org/10.1111/cgf.12129).
- MADDIGAN, Paula; SUSNJAK, Teo. Chat2Vis: Generating Data Visualizations via Natural Language using ChatGPT, Codex and GPT-3 Large Language Models. **IEEE Access**, v. 11, p. 45181–45193, 2023. DOI: [10.1109/ACCESS.2023.3273215](https://doi.org/10.1109/ACCESS.2023.3273215).
- MALLICK, Tanwi *et al.* ChatVis: Automating Scientific Visualization with a Large Language Model. *In: PROC. SC-W. [S. l.: s. n.]*, 2024. p. 49–55. DOI: [10.1109/SCW63240.2024.00014](https://doi.org/10.1109/SCW63240.2024.00014).
- MAPBOX GL JS. *[S. l.: s. n.]*, 2024. <https://www.mapbox.com/mapbox-gl-js>.
- MAPLIBRE GL JS. *[S. l.: s. n.]*, 2024. <https://maplibre.org/>.
- MCCABE, Thomas J. A Complexity Measure. **IEEE Trans. Softw. Eng.**, v. 4, p. 308–320, 1976. DOI: [10.1109/TSE.1976.233837](https://doi.org/10.1109/TSE.1976.233837).
- MIRANDA, Fabio; DORAISWAMY, Harish *et al.* Shadow Accrual Maps: Efficient Accumulation of City-scale Shadows Over Time. **IEEE Trans. Vis. Comput. Graph.**, v. 25, n. 3, p. 1559–1574, 2019. DOI: [10.1109/TVCG.2018.2802945](https://doi.org/10.1109/TVCG.2018.2802945).
- MIRANDA, Fabio; ORTNER, Thomas *et al.* The State of the Art in Visual Analytics for 3D Urban Data. **Comput. Graph. Forum**, v. 43, n. 3, e15112, 2024. DOI: [10.1111/cgf.15112](https://doi.org/10.1111/cgf.15112).
- MOREIRA, Gustavo; FERREIRA, Leonardo *et al.* Urbanite: A Dataflow-Based Framework for Human-AI Interactive Alignment in Urban Visual Analytics. **IEEE Trans. Vis. Comput. Graph.**, v. 32, n. 1, p. 1065–1075, 2026. DOI: [10.1109/TVCG.2025.3634644](https://doi.org/10.1109/TVCG.2025.3634644).
- MOREIRA, Gustavo; HOSSEINI, Maryam; NIPU, Md Nafiul Alam *et al.* The Urban Toolkit: A Grammar-based Framework for Urban Visual Analytics. **IEEE Trans. Vis. Comput. Graph.**, v. 30, n. 1, p. 1402–1412, 2023. DOI: [10.1109/TVCG.2023.3326922](https://doi.org/10.1109/TVCG.2023.3326922).

MOREIRA, Gustavo; HOSSEINI, Maryam; VEIGA, Carolina *et al.* Curio: A Dataflow-Based Framework for Collaborative Urban Visual Analytics. **IEEE Trans. Vis. Comput. Graph.**, v. 31, n. 1, p. 1224–1234, 2025. DOI: [10.1109/TVCG.2024.3456353](https://doi.org/10.1109/TVCG.2024.3456353).

MORITZ, Dominik *et al.* Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. **IEEE Trans. Vis. Comput. Graph.**, v. 25, n. 1, p. 438–448, 2018. DOI: [10.1109/TVCG.2018.2865240](https://doi.org/10.1109/TVCG.2018.2865240).

MOYROUD, Nicolas; PORTET, Frédéric. Introduction to QGIS. **QGIS and Generic Tools**, v. 1, p. 1–17, 2018. DOI: [10.1002/9781119439165.ch1](https://doi.org/10.1002/9781119439165.ch1).

MUNZNER, Tamara. A Nested Model for Visualization Design and Validation. **IEEE Trans. Vis. Comput. Graph.**, v. 15, n. 6, p. 921–928, 2009. DOI: [10.1109/TVCG.2009.111](https://doi.org/10.1109/TVCG.2009.111).

PARTL, Christian *et al.* enRoute: Dynamic Path Extraction from Biological Pathway Maps for Exploring Heterogeneous Experimental Datasets. **BMC Bioinformatics**, v. 14, S19, s3, 2013. DOI: [10.1186/1471-2105-14-S19-S3](https://doi.org/10.1186/1471-2105-14-S19-S3).

RAASVELDT, Mark; MÜHLEISEN, Hannes. DuckDB: An Embeddable Analytical Database. *In: PROC. SIGMOD*. [S. l.: s. n.], 2019. p. 1981–1984. DOI: [10.1145/3299869.3320212](https://doi.org/10.1145/3299869.3320212).

RODGERS, Johnny; BARTRAM, Lyn. Exploring Ambient and Artistic Visualization for Residential Energy Use Feedback. **IEEE Trans. Vis. Comput. Graph.**, v. 17, n. 12, p. 2489–2497, 2011. DOI: [10.1109/TVCG.2011.196](https://doi.org/10.1109/TVCG.2011.196).

RULFF, Joao *et al.* Urban Rhapsody: Large-scale Exploration of Urban Soundscapes. *In: COMPUT. Graph. Forum*. [S. l.: s. n.], 2022. v. 41, p. 209–221. DOI: [10.1111/cgf.14533](https://doi.org/10.1111/cgf.14533).

SATYANARAYAN, Arvind *et al.* Vega-Lite: A Grammar of Interactive Graphics. **IEEE Trans. Vis. Comput. Graph.**, v. 23, n. 1, p. 341–350, 2016. DOI: [10.1109/TVCG.2016.2599030](https://doi.org/10.1109/TVCG.2016.2599030).

SEDLMAIR, Michael; MEYER, Miriah; MUNZNER, Tamara. Design Study Methodology: Reflections from the Trenches and the Stacks. **IEEE Trans. Vis. Comput. Graph.**, v. 18, n. 12, p. 2431–2440, 2012. DOI: [10.1109/TVCG.2012.213](https://doi.org/10.1109/TVCG.2012.213).

STENGEL-ESKIN, Elias; PRASAD, Archiki; BANSAL, Mohit. Regal: Refactoring Programs to Discover Generalizable Abstractions. **arXiv preprint arXiv:2401.16467**, 2024. DOI: [10.48550/arXiv.2401.16467](https://doi.org/10.48550/arXiv.2401.16467).

WAGNER, Jorge *et al.* Reimagining TaxiVis through an Immersive Space-Time Cube Metaphor. *In: PROC. VR.* [*S. l.: s. n.*], 2024. p. 827–838. DOI: [10.1109/VR58804.2024.00099](https://doi.org/10.1109/VR58804.2024.00099).

WONGSUPHASAWAT, Krist. Encodable: Configurable Grammar for Visualization Components. *In: PROC. VIS.* [*S. l.: s. n.*], 2020. p. 131–135. DOI: [10.1109/VIS47514.2020.00034](https://doi.org/10.1109/VIS47514.2020.00034).

YU, Bowen; SILVA, Cláudio T. FlowSense: A Natural Language Interface for Visual Data Exploration Within a Dataflow System. **IEEE Trans. Vis. Comput. Graph.**, v. 26, n. 1, p. 1–11, 2019. DOI: [10.1109/TVCG.2019.2934668](https://doi.org/10.1109/TVCG.2019.2934668).

ZHANG, Ran; ELHAMOD, Mohannad. Data-to-Dashboard: Multi-agent LLM Framework for Insightful Visualization in Enterprise Analytics. **arXiv preprint arXiv:2505.23695**, 2025. DOI: [10.48550/arXiv.2505.23695](https://doi.org/10.48550/arXiv.2505.23695).

ZHAO, Yuheng *et al.* LightVA: Lightweight Visual Analytics with LLM Agent-based Task Planning and Execution. **IEEE Trans. Vis. Comput. Graph.**, v. 31, n. 9, p. 6162–6177, 2024. DOI: [10.1109/TVCG.2024.3413998](https://doi.org/10.1109/TVCG.2024.3413998).

ZIEGLER, Parker; CHASINS, Sarah E. A Need-finding Study with Users of Geospatial Data. *In: PROC. CHI.* [*S. l.: s. n.*], 2023. p. 1–16. DOI: [10.1145/3544548.3581370](https://doi.org/10.1145/3544548.3581370).