QuASAr: A Declarative Architecture for Adaptive Quantum Circuit Simulation

Tim Littau
t.m.littau@tudelft.nl
Delft University of Technology
Delft, Netherlands

Abstract

Quantum circuit simulation is essential for quantum algorithm development and verification, especially in the current stage of quantum computing where hardware is still largely error-prone and noisy. However, selecting the optimal simulation method for a given circuit is not trivial and can vary greatly from circuit to circuit. Current quantum simulation frameworks already offer automatic selection of simulation methods, but do so in a very naive way, often leading to suboptimal performance. Drawing inspiration from declarative database systems, we propose QuASAr (Quantum Adaptive Simulation Architecture), a declarative system for quantum circuit simulation that automatically analyses circuit characteristics, partitions circuits based on them and selects optimal simulation methods for each partition. QuASAr employs a novel two-level optimisation strategy: method-based partitioning groups circuit sections by optimal simulation technique, followed by entanglement-aware parallelism optimisation within each partition.

Keywords

Quantum Data Management, RDBMS, Simulation, Declarative Programming

ACM Reference Format:
Tim Littau. 2026. QuASAr: A Declarative Architecture for Adaptive Quantum Circuit Simulation. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Quantum computing has emerged as a transformative force in the evolution of computing technology, promising computational capabilities far beyond what traditional computers can achieve in previously virtually unsolvable problems [18]. However, quantum computing is still in a nascent stage, i.e., the noisy intermediate-scale quantum (NISQ) era, characterised by quantum computers that are constrained by noise and limited numbers of qubits [18]. For instance, IBM's Heron processor, one of the most performant quantum processors, has only 156 qubits and lacks full error correction. In this landscape, quantum circuit simulation serves as a core mechanism for exploring and validating quantum technologies, advancing quantum algorithm design, hardware development, and error correction [5].

Quantum circuit simulation refers to the use of classical computers to numerically reproduce the behaviour of quantum circuits, tracking the evolution of quantum states and operations. Existing simulation frameworks such as Qiskit Aer [2], Cirq's qsim [1], and specialized simulators like Stim [11] are typically limited to pick one simulation technique for a provided circuit. Each method has distinct performance characteristics and applicability domains: statevector simulation provides universal compatibility but scales exponentially with qubit count; stabilizer tableau methods [3] are efficient for Clifford circuits but limited in scope; decision diagrams [25] can be memory-efficient for certain circuit structures but perform poorly on dense entanglement patterns; matrix product states [21] handle moderate entanglement efficiently but struggle with highly entangled states.

This singular method selection creates several fundamental challenges in the NISQ era:

(1) **Expertise Barrier**: Users must understand the nuances of different simulation techniques and their performance trade-offs if they want to manually partition a circuit and give it to a system piece by piece
(2) **Suboptimal Performance**: This selection often leads to poor performance choices, as users may not have comprehensive knowledge of all available methods
(3) **Resource Underutilization**: Existing systems fail to exploit parallelism opportunities within and across simulation methods
(4) **Scalability Limitations**: No unified approach exists for handling large, heterogeneous quantum circuits that could benefit from different simulation methods for different sections

Drawing inspiration from declarative database systems that abstract query execution details from users through automatic query optimization, we propose QuASAr (Quantum Adaptive Simulation Architecture), a declarative system for quantum circuit simulation. Just as database query optimizers automatically choose between different join algorithms, access methods, and execution strategies based on data characteristics and system resources, QuASAr automatically analyzes circuit characteristics, partitions circuits based on method heuristics and parallelism, and selects optimal simulation methods for each partition.

Prior visioning work identified two central questions for the NISQ-era simulation stack: (i) the need for principled optimizers that select or combine simulation methods per circuit region, and (ii) the search for compressed, compositional state representations that

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

admit efficient conversion between methods. [13] QuASAr directly operationalizes these directions by combining a theory-first cost model, a compact interface descriptor (SSD), and switching-aware planning.

1.1 Contributions

This paper makes the following key contributions:

(1) **Declarative Quantum Simulation API**: We introduce a declarative interface for quantum circuit simulation that abstracts method selection and circuit partitioning complexity from users, similar to how SQL abstracts query execution in database systems.
(2) **Two-Level Optimization Framework**: We develop a novel optimization strategy that combines method-based circuit partitioning with entanglement- and gate-type-aware parallelism optimization, maximizing performance across diverse circuit types.
(3) **Adaptive Method Selection**: We present heuristic algorithms for automatic simulation method selection based on circuit structural heuristics.
(4) **Comprehensive Evaluation**: We provide extensive experimental evaluation on commonly used quantum circuit benchmarks.
(5) **Unified Backend Integration**: We design a modular architecture that seamlessly integrates multiple simulation backends (statevector, stabilizer, decision diagram, tensor networks) through a unified interface while remaining easily extendable.

1.2 Outline

The remainder of this paper is organized as follows. Section 2 provides background on quantum circuit simulation methods and their characteristics. Section 3 presents the QuASAr system architecture and declarative API. Section 4 details our multi-level optimization algorithms. Section 6 describes the system implementation. Section 8 presents experimental results. Section 7 discusses related work, and Section 9 concludes the paper.

2 Background and Motivation

2.1 Quantum Computing Fundamentals

To understand the data management challenges in quantum circuit simulation, we first establish the fundamental concepts of quantum computing that differentiate it from classical computing.

Quantum Data vs. Classical Data. Classical data consists of bits that are either 0 or 1. In contrast, quantum data is represented by quantum bits (qubits), which can exist in superposition—a linear combination of both 0 and 1 states simultaneously. Mathematically, a single qubit state $|\psi\rangle$ can be represented as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex amplitudes satisfying $|\alpha|^2 + |\beta|^2 = 1$. When measured, the qubit collapses to state $|0\rangle$ with probability $|\alpha|^2$ or state $|1\rangle$ with probability $|\beta|^2$.

For an n-qubit quantum system, the state space grows exponentially to 2^n dimensions, requiring a complex vector of size 2^n to represent the complete quantum state. This exponential scaling is one of the sources of both quantum computing's potential power and the computational challenges in classical simulation.

Quantum Circuits and Gates. Quantum circuits consist of sequences of quantum gates applied to qubits. Gates can be categorized into several types based on their computational properties:

- **Clifford Gates**: Include Hadamard (H), Pauli gates (X, Y, Z), and the controlled versions of these gates. These gates preserve the stabilizer structure and can be simulated efficiently on classical computers using polynomial resources.
- **Non-Clifford Gates**: Include T gates and arbitrary rotation gates. These gates enable universal quantum computation but require exponential resources for classical simulation.
- **Entangling Gates**: Such as controlled gates (e.g. CX, CY, CZ), create quantum entanglement between qubits, correlating their measurement outcomes.

Quantum Entanglement. A crucial quantum phenomenon and the other source of quantum computing's power where multiple qubits become correlated such that measuring one qubit immediately affects the others, regardless of physical distance. Entanglement patterns significantly impact simulation complexity and method selection.

2.2 Quantum Circuit Simulation Methods

The diversity of quantum simulation methods reflects different approaches to managing the exponential complexity of quantum systems, here we present some of the most widely used methods:

Statevector Simulation explicitly represents the complete quantum state as a complex vector of size 2^n. This method provides universal compatibility with all quantum gates but suffers from exponential memory scaling, limiting practical use to approximately 30 qubits on commodity hardware. The computational complexity for applying a gate is $O(2^n)$.

Stabilizer Tableau Simulation exploits the mathematical structure of Clifford circuits by representing quantum states using stabilizer generators rather than explicit state vectors. This approach scales polynomially in qubit count ($O(n^2)$ space and $O(n^2)$ time per gate) but is restricted to Clifford gates, making it unsuitable for universal quantum computation. It is commonly used in simulating quantum error correction codes [11].

Decision Diagram (DD) Simulation represents quantum states and operations using decision diagrams, which can provide exponential compression for circuits with specific structural properties. DD methods excel with sparse entanglement patterns and structured circuits but degrade significantly for densely entangled circuits. The performance depends heavily on the circuit's entanglement structure.

Tensor Network (TN) Simulation represents quantum states as interconnected tensors rather than explicit state vectors. By decomposing the 2^n-dimensional state space into a network of smaller tensors with controlled bond dimensions, TN methods can efficiently capture states with limited entanglement. The computational complexity depends on the network structure and bond dimension χ, and can scale polynomially with system size for circuits with low entanglement. TN approaches generalize multiple representations, such as Matrix Product States (MPS), Projected Entangled Pair States (PEPS), and Tree Tensor Networks (TTN). They

are highly effective for simulating circuits with local interactions and shallow depth, but their efficiency deteriorates in the presence of volume-law entanglement, which is typical in many quantum algorithms.

Matrix Product State (MPS) Simulation is a commonly used variant of TN and represents quantum states as products of matrices with controlled bond dimension χ. The computational complexity scales as $O(n\chi^3)$, where χ controls the trade-off between accuracy and computational cost. MPS methods handle moderate entanglement efficiently but struggle with highly entangled states common in quantum algorithms.

2.3 The Method Selection Problem

The diversity of simulation methods creates a complex optimization problem analogous to query optimization in database systems. Consider a quantum circuit implementing Shor's algorithm: the initial quantum Fourier transform sections exhibit structured entanglement patterns well-suited for DD simulation, while the modular exponentiation sections may benefit from statevector simulation for small instances or MPS for larger ones. The arithmetic sections might be efficiently handled using stabilizer methods if they consist primarily of Clifford gates.

Current practice in quantum circuit simulation typically applies a single simulation method to the entire circuit, even when the circuit contains regions that could be more efficiently simulated using different approaches. For example, applying statevector simulation to a large Clifford-dominated region wastes computational resources, while using stabilizer methods for segments containing non-Clifford gates is insufficient or approximated at best. Although some frameworks, such as Qiskit Aer, automate the choice of simulation method, the selection remains singular and does not exploit potential efficiency gains from partitioning the circuit and applying different methods to different regions. This limitation becomes particularly pronounced in heterogeneous circuits, where the optimal simulation strategy varies across subcircuits. The challenge is compounded by the fact that optimal method selection depends on multiple factors:

- **Gate Composition**: The ratio of Clifford to non-Clifford gates
- **Entanglement Structure**: Sparse vs. dense, local vs. global entanglement patterns
- **Circuit Size**: Number of qubits and gate depth
- **Resource Constraints**: Available memory and computational time
- **Accuracy Requirements**: Tolerance for approximation errors

2.4 Parallelism Opportunities in Quantum Simulation

Quantum circuit simulation presents multiple parallelism opportunities that existing systems fail to exploit systematically:

Method-Level Parallelism: Different circuit sections can be simulated using different methods simultaneously, with results combined through tensor products or state vector operations.

Entanglement-Aware Parallelism: Circuit sections with independent qubit groups can be simulated in parallel before entangling operations merge the subsystems [19]. This requires careful analysis of the circuit's entanglement graph to identify independent components.

Temporal Parallelism: Gates operating on disjoint qubit sets or commutative gates (e.g. phase gates) can be executed simultaneously within the same simulation method, similar to instruction-level parallelism in classical processors.

Data Parallelism: For methods like statevector simulation, the state vector can be partitioned across multiple processors, enabling parallel gate applications [10].

Table 1: Analogy Between Database Query Processing and Circuit Simulation Strategy

Database Concept	Circuit Simulation
Declarative Query	Quantum Circuit
Physical Data Layout *(e.g., Columnar, Row-Store)*	State Representation *(e.g., Tableau, DD, State Vector)*
Query Optimizer	Circuit Partitioner & Estimator
Catalog Statistics	Circuit Structure Metrics
Physical Query Plan	Partitioned Simulation Plan
Physical Operators	Simulation Backends
Data Conversion Cost (ETL)	Representation Conversion Cost

2.5 Database Systems Analogy

The quantum simulation method selection problem closely parallels query optimization in database systems. Table 1 illustrates this correspondence:

Just as database query optimizers automatically choose between nested loop joins, hash joins, and sort-merge joins based on table sizes, selectivity, and available indexes, quantum circuit simulation systems should automatically select between statevector, stabilizer, DD, and TN methods based on circuit characteristics and computational constraints.

This analogy motivates our declarative approach: users specify *what* to simulate (the quantum circuit) rather than *how* to simulate it (the specific methods and resource allocation). The system takes responsibility for analyzing the circuit, estimating costs, and selecting optimal execution strategies—just as SQL databases handle query optimization transparently.

3 QuASAr System Architecture

3.1 Declarative API Design

QuASAr provides a simple, declarative interface that abstracts simulation complexity from users, similar to how SQL abstracts query execution details in database systems:

```
result = quasar.simulate(circuit,
                         target_accuracy=0.99,
                         max_time=300,
                         max_memory="16GB",
                        optimization_level="aggressive")
```

This interface accepts a quantum circuit specification and optional performance constraints, returning simulation results without requiring users to understand underlying simulation methods. The system automatically handles method selection, partitioning, and resource management. Users can specify constraints such as accuracy requirements, time limits, memory bounds, and optimization preferences, allowing the system to make informed trade-offs during execution.

3.2 System Components

Figure 1 illustrates QuASAr's architecture, consisting of five main components organized in a layered approach:

Circuit Analyzer examines input circuits to extract structural characteristics including gate types, entanglement patterns, temporal locality, and resource requirements. The analyzer computes multiple metrics:

- **Gate Distribution**: Counts of Clifford vs. non-Clifford gates, rotation angles, and respective depths
- **Entanglement Analysis**: Connectivity graphs, clustering coefficients, and component analysis
- **Temporal Structure**: Gate dependencies, parallelizable sections, and critical path analysis
- **Resource Estimation**: Memory requirements and computational complexity for different methods

Method Selector uses circuit analysis results to determine optimal simulation methods for different circuit sections. The selector employs a multi-criteria decision framework that considers:

- Method compatibility scores based on gate types and circuit structure
- Performance models based on heuristics and estimations
- Resource availability and constraints
- User-specified accuracy and time requirements

Partition Optimizer implements our two-level partitioning strategy, first creating method-based partitions, then optimizing each partition for parallel execution. The optimizer uses graph algorithms to identify optimal cut points and balance computational load across partitions.

Execution Engine coordinates multiple simulation backends, manages resource allocation, and handles inter-partition communication and state synchronization. The engine implements a sophisticated scheduling system that maximizes resource utilization while respecting dependencies between partitions.

Structural State Decomposition (SSD) as an overarching hierarchical data structure representing the partitioned circuit while preserving compression within partitions. This way the output of a simulation will never manifest the full statevector if unfeasible, avoiding the loss of any performance gain. Instead, SSD delivers the output in its native representation. If a circuit is entirely simulated using Tableau representation, it is output as Tableau, if it is a mixture of Tableau and Decision Diagram simulation for different qubit subsystems each subsystem is output using its respective representation.

3.3 SSD in detail

QuASAr represents quantum circuits using an enhanced directed acyclic graph (DAG) structure that extends traditional gate-based representations with additional metadata for optimization:

- **Circuit Partition**: Represent the partition of the parent circuit with its individual quantum operations, parameters, target qubits, and compatibility annotations
- **Dependency Edges**: Encode temporal dependencies between gates based on qubit usage and entanglement creation
- **Entanglement Annotations**: Track which gates create or modify entanglement between qubit groups
- **Method Compatibility Labels**: Indicate which simulation methods can handle each gate type efficiently
- **Resource Metadata**: Store computational and memory requirement estimation for each circuit segment

The SSD design is informed by recent visioning work that stresses the importance of succinct, compositional representations for simulation and conversion (so as to avoid full state materialization). [13] SSD intentionally summarizes only the boundary qubits, rank bounds and frontier statistics, enabling partial conversion strategies (Section 5).
SSD enables efficient analysis of circuit structure and supports the partitioning algorithms described in Section 4. The enhanced DAG allows for rapid identification of parallelizable sections, optimal partition boundaries and ideal conversion layers.

3.4 Backend Integration

QuASAr integrates with multiple simulation backends through a unified adapter interface that provides method-specific optimizations while maintaining consistency and remaining modular. Current backends within QuASAr include:

- **Statevector Backend**: Qiskit Aer for statevector simulation with GPU acceleration support using the cuStateVec backend if supported
- **Stabilizer Backend**: Stim for efficient Clifford circuit simulation with optimized tableau operations
- **Decision Diagram Backend**: Custom DD implementation optimized for sparse circuits with dynamic node ordering
- **MPS Backend**: Qiskit Aer for matrix product state simulation with variable bond-dimension selection

Each backend adapter handles method-specific optimizations such as gate fusion, memory management, and parallel execution while providing a consistent interface to the execution engine.

4 Multi-Level Optimization Algorithms

The optimizer in QuASAr is a multi-level, cost-based planner that chooses — at runtime — the best simulation backend for each circuit fragment. Its design mirrors classical query optimizers like System-R style Dynamic Programming (DP) for join ordering, adaptive re-optimization, and materialization tradeoffs applying familiar reasoning and tooling to simulation scheduling from database research [4, 12, 20]. The high-level call for a dedicated simulation optimizer (e.g., to reason about contraction order, sparsity, and noise-aware policies) appears in prior visioning work [13]; QuASAr

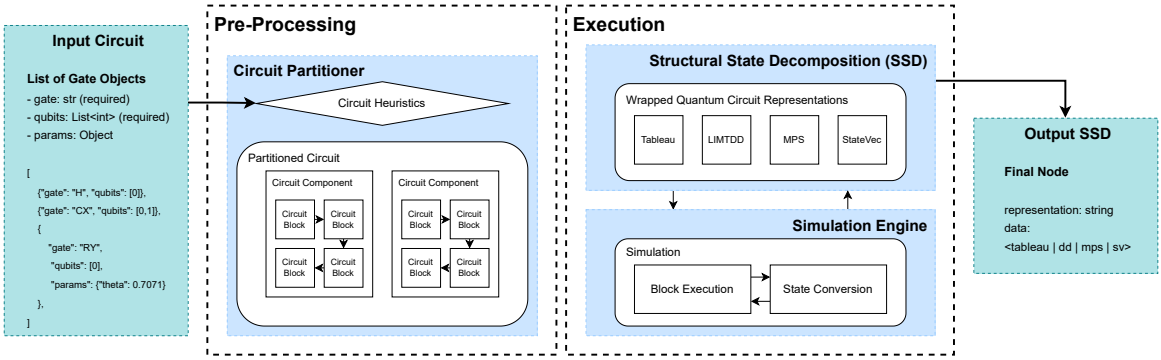

Figure 1: QuASAr system architecture showing the flow from circuit input through analysis, optimization, and execution phases.

answers this call with a switching-aware DP planner that jointly optimizes per-partition backend choice and conversion penalties.

4.1 Problem statement and notation

Let $G = (g_1, \ldots, g_m)$ be the gate sequence of a quantum circuit. We choose an ordered set of candidate cut positions that partition G into contiguous *partitions* $p_1, \ldots, p_c$. For each partition p and simulation backend b (from the finite set $\mathcal{B} = \{\text{SV, DD, MPS, Tab}\}$) we require compact, efficiently-evaluable cost estimates:

- $C(p, b)$: total estimated cost to simulate partition p on backend b. This is a scalar combining CPU time with a weighted memory term; each C is assembled from theoretical building blocks (e.g., $O(\chi^3)$ dependence for MPS SVD) and microbench-calibrated constants.
- $S(b' \to b, I)$: switching cost to convert an interface produced by backend b' into a representation usable by backend b. The descriptor I is a compact SSD for the cut (boundary qubits, rank bound, frontier size, fingerprint).
- $\text{best}[j, b]$: the minimal cumulative cost to execute partitions $p_1..p_j$ and end with partition p_j simulated on backend b. This is the DP state.
- $\text{Gain}(p \to p_a, p_b)$: an estimated saving for splitting partition p into p_a, p_b (used in coarse-to-fine refinement).
- $\beta : \{1, \ldots, c\} \to \mathcal{B}$: backend assignment function mapping partition index to a choosen backend

With these primitives the global planning objective is:

$$\min_{\beta:\{1,\ldots,c\}\to\mathcal{B}} \sum_{i=1}^{c} C(p_i, b_i) \; + \; \sum_{i=1}^{c-1} S(b_i \to b_{i+1}, I_{i,i+1}).$$

Intuitively: pick a backend for each partition so simulation plus conversion costs are minimized. The optimizer supports both offline full-circuit planning and online sliding-window planning; the DP below is the core engine used in both modes.

4.2 DP planner: intuition and recurrence

We build optimal prefix solutions and extend them partition by partition. The DP state $\text{best}[j, b]$ represents the cheapest way to finish partitions $1..j$ with the last partition p_j executed by b. To extend a prefix we choose an earlier cut $i < j$ (so the new last partition is $p_{i+1..j}$) and a previous backend b' used for the prefix; the cost to join these two pieces is $\text{best}[i, b'] + C(p_{i+1..j}, b) + S(b' \to b, I)$.

Boxed recurrence (compact):

$$\text{best}[j, b] = \min_{0 \le i < j} \min_{b' \in \mathcal{B}} \left(\text{best}[i, b'] + C(p_{i+1..j}, b) + S(b' \to b, I_{i,j}) \right)$$

Base: $\text{best}[0, b] = 0$ represents the empty prefix. The optimal overall plan is $\min_b \text{best}[c, b]$; backpointers recorded during DP permit exact reconstruction of the chosen cuts and backends.

4.3 Algorithm and practical notes

The following Algorithm shows the planner in compact, commented pseudocode. It is intentionally simple: the key engineering levers (pruning, beam width, SSD memoization) are described after the algorithm.

Algorithm 1 DP planner for contiguous partitions.

Require: Partitions $p_1..p_c$, backends $\mathcal{B}$, cost functions C, S

```
Initialize best[0, b] ← 0 and best[j, b] ← ∞ for j ≥ 1, all b
for j = 1 to c do                                   ▷ fill DP table left-to-right
    for each backend b ∈ B do
        for each split index i with 0 ≤ i < j do
            q ← p_{i+1..j}
            I_i
            for each previous backend b' ∈ B do
                cand ← best[i, b'] + C(q, b) + S(b' → b, I)
                if cand < best[j, b] then
                    best[j, b] ← cand; (i, b') recorded
                end if
            end for
        end for
    end for
end for
return min_b best[c, b] and the reconstructed plan
```

Backpointer recovery. For each updated $\text{best}[j, b]$ we store the argmin (i^*, b'^*). Recover the chosen segmentation and backends by tracing back from (c, b^*) following recorded backpointers to $j = 0$.

Connection to System-R. System-R enumerates optimal plans for join subsets by DP over subsets; our DP is analogous but specialized to contiguous partitions (the linear order of gates), so complexity and state representation differ but the principle (optimal-substructure, memoization of subplans) is the same [12, 20].

4.4 Granularity and coarse-to-fine refinement

The planner requires a candidate partition set. Choosing the partition density is the key trade-off:

- Fine-grained (local groups): very expressive, expensive DP.
- Coarse-grained (logical blocks): cheap DP, may miss local gains.
- Practical compromise — hierarchical refinement: start with coarse partitions; compute a conservative Gain$(p \to p_a, p_b)$ estimate and split only those partitions where Gain $< -\tau$. Re-run or incrementally update DP locally.

This keeps the planner's c small while enabling fine-grained switching where it is beneficial.

4.5 Multi-objective planning and pruning

When optimizing time and memory separately, each best$[j, b]$ holds a small Pareto frontier. To keep state manageable we combine:

- **Epsilon-dominance**: merge near-equal vectors.
- **Size cap** K: keep top-K nondominated candidates per state.
- **Branch-and-bound**: use optimistic lower bounds to prune early.

4.6 Practical heuristics and complexity

We employ practical heuristics used in DB optimizers:

- **Beam / top-K**: limit per-state candidates.
- **SSD coalescing and memoization**: group similar SSDs and cache S.
- **Sliding horizon**: run DP only over a window of upcoming partitions during online execution.

Complexity summary: with c candidate partitions and $|\mathcal{B}|$ backends the scalar DP performs $O(c^2 \cdot |\mathcal{B}|^2)$ cost evaluations. With top-K pruning and SSD coalescing the practical cost is significantly lower.

4.7 Adaptive re-optimization

Execution is instrumented: observed times and conversion durations update calibration constants (exponential moving averages). Re-optimization is triggered when drift exceeds threshold β or when memory quotas approach limits. Memoized subplans accelerate replanning.

5 Advanced Optimization Techniques

This section details the optimization machinery that enables cheap, partial transitions between backends and supports adaptive partitioning. It uses the compact notation from Section 4: partitions p_i, cost primitives $C(p, b)$, switching cost $S(b' \to b, I)$, and SSD interface fields $I = (Q, s, r, \text{FP})$ (boundary qubits Q, rank bound s, DD frontier size r, fingerprint FP). The presentation emphasizes concise algorithms, cost-aware decision rules, and pragmatic safety limits.

5.1 SSD: concise semantics and guarantees

An SSD (Structural State Decomposition) is a compact descriptor of the state at a cut: $I = (Q, s, r, \text{FP})$. Important properties:

- **Compactness:** SSD is *not* a full statevector. It stores small canonical objects (ordered boundary qubit list Q, a rank bound s, a frontier-size estimate r, and a short structural fingerprint FP). SSD size is $O(|Q| \cdot s)$ or $O(\text{poly}(s, |Q|))$, typically tiny compared to $2^{|Q|}$.
- **Conservative bounds:** s is an upper bound on the Schmidt rank (or effective bond dimension) across the cut; r upper-bounds frontier complexity in DDs. If any bound is unknown or exceeds configured safety caps (e.g., $s > s_{\max}$ or $r > r_{\max}$) the optimizer treats switching as expensive.
- **Reusability:** SSDs and small bridge tensors can be cached (keyed by FP) and reused across repeated or similar partitions, enabling zero- or low-cost re-use of prior conversions.

Because SSDs avoid full materialization they preserve compression and make partial conversions feasible in practice.

5.2 Conversion primitives

We implement three layered conversion primitives. Each primitive aims to keep work proportional to SSD size rather than to a full $2^{|Q|}$ state.

1. Boundary-to-boundary (B2B). Extract a small bridge tensor B on qubits Q, truncated to rank s, and ingest B into the target backend. Costs (sketch):

$$E_{\text{ext}}(b', I) + T_{\text{copy}}(|B|) + I_{\text{ingest}}(b, I),$$

where E_{ext} uses s (e.g., $O(s^3)$ SVD work for MPS), $T_{\text{copy}} \propto |B|/bw$, and I_{ingest} depends on target. B2B is preferred when s and $|Q|$ are small.

2. Local-window extraction (LW). Extract a dense state on a local window of w qubits (with w chosen so 2^w is affordable), then convert that dense window into the target representation (truncated SVD → MPS, insert → DD, stabilizer learning → tableau). Cost dominated by $O(2^w)$ arithmetic plus ingestion.

3. Staged / hybrid conversion (ST). Perform a staged pipeline: first approximate the state via a compact intermediate (e.g., low-χ MPS), then progressively refine / reproject into the target backend only as needed. Staging amortizes SVD/contract costs and permits incremental commitment if the DP decides to continue with the new backend.

5.3 Conversion strategy summary

Table 2 summarizes conversion strategies, typical applicability, and dominant cost terms (expressed using SSD fields). These are cost *sketches*—actual numeric costs use microbench-calibrated constants (Section 4).

5.4 Algorithmic flow for switching

Given a planned switch $b' \to b$ at cut with SSD I, the conversion engine executes:

(1) **Lookup cache:** check FP-keyed cache for an existing bridge tensor or measured $S(b' \to b, I)$. If hit and bridge is valid, skip extraction.

Table 2: Conversion primitives (sketch).

ID	When used	Dominant cost (big-O sketch)
B2B	small rank s, small $\|Q\|$; both sides accept boundary injection	$O(s^3)$ (SVD) + $O(\|B\|/bw)$ (copy) + ingest_cost(s)
LW	boundary large but small local window w exists	$O(2^w)$ (dense extraction) + ingestion cost
ST	s moderate/large; try low-χ approximate bridge then refine	$O(\tilde{\chi}^3)$ (approx SVD) then incremental refine; amortized by reuse
Full	fallback: full statevector extraction (expensive)	$O(2^{\|Q\|})$ — avoided when possible

(2) **Feasibility check:** if $s > s_{\max}$ or $r > r_{\max}$, abort with infeasible (treat S as very large). Else continue.
(3) **Attempt B2B:** if s and $|Q|$ permit, compute bridge via truncated SVD or frontier projection; validate reconstruction error (norm loss $\leq \epsilon$). If success, transfer and ingest.
(4) **Fallback LW:** if B2B fails or reconstruction poor, attempt LW with window width w (heuristic choose smallest w s.t. extraction cost acceptable).
(5) **Staging / refine:** if LW still too expensive, produce a staged low-$\tilde{\chi}$ approximation and inject; refine on demand.
(6) **Post-check:** after ingestion, validate target's internal diagnostics (e.g., MPS bond growth). If ingestion exceeded budget, roll back to prior backend and mark cut as 'no-switch' for future plans unless policy overrides.

5.5 Cost-aware heuristics and safety limits

To ensure robust operation we enforce conservative safety policies:

- $\mathbf{s}_{\max}$: maximum allowed rank for B2B extraction; default tuned to keep SVD cheap.
- $\mathbf{r}_{\max}$: frontier visit cap for DD traversal; if exceeded treat 'r=' (infeasible).
- $\mathbf{w}_{\max}$: maximum local-window width for LW (practical cap like $w_{\max} \leq 24$).
- τ: split-gain threshold for hierarchical refinement (see Section 4).

These thresholds are exposed in configuration and are part of the DP's conservative policy so that planner avoids pathological, worst-case conversions.

5.6 Caching and reuse

Fingerprints FP enable aggressive caching:

- **Bridge cache:** map FP $\mapsto$ (bridge tensor, target-ready handle, measured S). If reused, conversion cost reduces to cache lookup + optional validation.
- **Cost cache:** if multiple DP branches consider the same SSD, reuse previously computed $S(\cdot)$.
- **Delta updates:** if a cached bridge with rank $\tilde{s}$ exists and new s differs slightly, attempt low-cost delta projection rather than full recompute.

Cached bridges must be stored with metadata: creation SSD, creation time, fidelity budget and memory footprint. A background eviction policy keeps cache size bounded.

5.7 Instrumentation and learning

Each conversion attempt records: SSD I, primitive used (B2B/LW/ST), time (breakdown by ext/copy/ingest), memory peak, and fidelity loss. These records feed two components:

- **Online calibration:** EMA updates of microbench constants (Section 4) to keep predicted S accurate.
- **Learned heuristics:** a lightweight classifier/ranker (trained from accumulated conversions) to predict cheap conversion routes or to quickly reject infeasible cuts; the classifier seeds DP but never replaces the theory-grounded cost model.

5.8 Fallbacks and robustness

Conversion can fail or exceed budget. QuASAr provides:

- **Graceful roll-back:** keep pre-switch checkpoints (SSD + small materialization) so we can roll back without re-running earlier fragments.
- **Conservative default:** if conversion uncertain, prefer staying in current backend for the fragment and mark the cut as 'deferred' for later re-evaluation.
- **Plan repair:** upon failure the online planner invokes a local DP to find a short-horizon repair plan that avoids costly fallback conversions.

5.9 Summary

Advanced optimization in QuASAr centers on SSD-driven partial conversions, a small palette of conversion primitives (B2B, LW, ST), and a cautious set of safety thresholds. SSDs preserve compression by avoiding full statevector materialization; combined with caching, staged conversion and conservative DP policies, this enables aggressive but robust multi-backend simulation for heterogeneous circuits.

6 Implementation

This section describes the concrete realization of QuASAr. Our goal is to provide sufficient implementation detail for reproducibility and evaluation while keeping the presentation concise and focused on algorithmic and systems-level design choices that matter to reviewers (rather than developer/build minutiae).

6.1 High-level architecture

QuASAr is implemented as a two-tier system (Figure 1): (i) a lightweight **control plane** implemented in Python that performs orchestration, plan search, and runtime scheduling; and (ii) a high-performance **conversion engine** implemented in C++ that performs the performance-critical numerical work (boundary extraction, SVDs, bridge-tensor construction, and stabilizer learning). The control plane interacts with existing, well-engineered simulators (Stim for tableau/stabilizer execution and the MQT Core DD package for decision-diagram execution) via their public APIs. Communication between the Python control plane and the C++ conversion engine uses concise, well-documented bindings (pybind11) and, wherever possible, zero-copy opaque handles to avoid unnecessary data movement.

6.2 Core components

The implementation is organized into a small set of modules whose responsibilities map directly to the optimizer and scheduler described in earlier sections:

Conversion engine (C++). This module contains:

- **Boundary extraction**: compute compact SSD descriptors and extract top-s Schmidt vectors or small frontier-DDs for fragment interfaces.
- **Local-window extraction**: produce dense windows (size bound by policy) for stabilizer learning or ingestion into other backends.
- **Stabilizer learner**: a robust routine that, given a dense window, attempts to recover stabilizer generators and to synthesize a tableau when possible.
- **Bridge management**: construction and application of bridge tensors that allow application of cross-fragment gates without full materialization.

These routines are written to be CPU-efficient and to expose cost estimates (time and memory) for their operations so the optimizer can account for conversion penalties precisely.

Optimizer & Scheduler (Python). The Python control plane implements:

- **Multi-level planner**: dynamic programming (DP) plan enumeration that minimizes combined simulation and switching cost, tracking SSD states at fragment boundaries.
- **Adaptive runtime manager**: monitors observed costs and triggers re-optimization when estimates deviate significantly or system constraints (memory quota) are at risk.
- **Back-end orchestrator**: invokes simulated code in Stim, MQT Core, or an MPS library using their established Python interfaces; orchestrates background conversions and plan prefetching.

Back-end adapters. Instead of reimplementing simulators, QuASAr integrates mature simulators via adapters that translate between QuASAr's SSDs/bridge objects and backend-specific state objects (Stim tableaus, MQT Core DD roots, or MPS tensors). The adapters encapsulate ownership, lifetime, and zero-copy rules for handles and provide a small, consistent API for the scheduler.

6.3 Interfaces and dataflow

The system exposes a concise API between the control plane and the conversion engine:

- `EstimateCost(fragment, backend)`: returns a cost tuple (T, M) computed from theoretical formulas plus calibrated micro-bench constants.
- `ExtractSSD(fragment, boundary_qubits, s)`: returns an SSD with optional bridge tensor truncated to top-s components.
- `ConvertBoundary(SSD, target_backend)`: performs a partial conversion (boundary or local-window) and returns a handle usable by the target backend.
- `TryBuildTableau(dense_vector)`: attempts stabilizer learning and, on success, returns a tableau object or a descriptive failure reason.

A typical dataflow for a planned backend switch is:

Scheduler → ExtractSSD → Conversion → inject into adapter

where the conversion step is performed in C++ and the scheduler remains free to continue other tasks. Conversions are cancellable and may be executed asynchronously to hide latency.

6.4 Design choices and rationale

Several deliberate engineering choices guided the implementation:

- **C++ conversion kernel, Python control plane.** Performance-critical numeric operations (SVDs, small dense linear algebra, frontier traversals) are implemented in C++ to leverage optimized BLAS/LAPACK and control memory precisely. The Python control plane enables rapid experimentation, easy integration with existing simulator Python APIs, and a succinct implementation of DP-based planning logic. This separation mirrors contemporary systems that combine a high-performance kernel with a flexible orchestration layer.
- **Leverage best-of-breed simulators.** Stim and the MQT Core DD package are used as execution engines; QuASAr focuses on scheduling and conversion. This reduces reinvention, leverages existing optimizations, and allows us to concentrate on scheduling and conversion correctness.
- **Theory-grounded costing with calibration.** Cost formulas are derived from algorithmic complexity (e.g., $O(\chi^3)$ for SVD steps in MPS, $O(\#\text{DD nodes})$ for DD operations) and are calibrated by microbenchmarks to produce realistic estimates across machines.

6.5 Runtime behavior and optimization loop

At runtime QuASAr runs the following loop:

(1) Fragment the input circuit using a coarse initial policy.
(2) For each fragment, compute theoretical cost estimates for all backends (using SSD descriptors where available).
(3) Run DP plan enumeration to obtain a candidate global plan.
(4) Execute the plan, invoking backends through their adapters. Perform planned conversions via the conversion engine.
(5) Instrument actual behavior (execution times, peak memory) and feed results back into the optimizer. If measurement shows deviations beyond a configured threshold, re-optimize over a sliding horizon.

6.6 Testing, benchmarks and evaluation setup

The implementation includes a benchmark harness and unit tests that focus on correctness (state fidelity) and performance (time / memory) of conversion primitives. For evaluation, we exercise QuASAr on a curated benchmark suite representative of database-like quantum workloads:

- **Clifford-dominant circuits**: error-correction and syndrome extraction subroutines (Tableau/Stim-friendly).
- **Structured arithmetic kernels**: adder/multiplier circuits exhibiting regularity (DD-friendly).
- **Variational circuits**: low-depth nearest-neighbor circuits with small entanglement growth (MPS-friendly).
- **Hybrid random circuits**: mixtures of Clifford and non-Clifford gates to stress switching logic.

We report wall-clock runtime, peak memory, conversion counts, and plan-cache hit rates. Microbenchmarks used to calibrate low-level constants are also supplied with the repository to allow reproducible cost-model tuning.

6.7 Reproducibility and artifacts

To facilitate reproducibility we provide:

- Source code for the conversion engine and adapters (C++), and the scheduler (Python).
- A benchmark harness and scripts to reproduce the evaluation runs in the paper.
- Microbenchmarks used for calibration and a recorded table of constants for the machines used in our experiments.

Implementation notes and developer-level build instructions are maintained in the project repository's README and a short appendix; these are intentionally separated from the paper to keep the manuscript focused and accessible to reviewers.

6.8 Limitations

The current implementation targets classical, CPU-based execution and focuses on conversion and scheduling algorithms rather than on accelerating backends (e.g., GPU-accelerated MPS). SSD extraction is intentionally conservative: when frontier sizes blow up the scheduler declines aggressive conversions and falls back to single-backend execution. These pragmatic choices improve robustness at the cost of not pursuing aggressive, potentially brittle approximations by default.

7 Related Work

QuASAr sits at the intersection of classical quantum-simulation methods and database-style cost-based optimization. In this section we situate QuASAr relative to the key strands of literature that informed its design: stabilizer/tableau simulation, decision-diagram based simulation, tensor-network / MPS-based simulation, hybrid and conversion-oriented representations, and multi-backend execution frameworks.

Prior visionary work has articulated research agendas for quantum data management in the NISQ era, highlighting the need for simulation optimizers and for new, compressible state representations that enable partial conversion and out-of-core execution. [13] QuASAr complements these agendas by providing an implementable design and algorithms for cost-based planning, SSD-driven partial conversion, and adaptive re-optimization.6)6)

7.1 Stabilizer / tableau simulators

Efficient simulation of stabilizer circuits (those composed of Clifford gates) is a classical result from the Gottesman–Knill line of work. Practical tableau-based implementations and optimizations were developed by Aaronson and Gottesman and later refined by the community; modern highly-optimized implementations such as Stim adopt these tableau ideas and scale them to very large stabilizer workloads (e.g., error-correction circuits) through algorithmic engineering and careful memory layout [3, 11]. QuASAr leverages tableau simulation as one of its native backends and uses tableau-dominance detection in its micro-level estimators.

7.2 Decision diagrams (DDs) and the MQT toolchain

Decision diagrams provide a graph-based compressed representation of quantum states and operators; their performance depends strongly on variable ordering and the structural regularity of circuits [24]. The MQT/DD ecosystem (the MQT Core / DD package) provides production-quality DD implementations and tooling used in many quantum EDA tasks [9]. QuASAr integrates DD-based cost models and extraction primitives (SSD frontiers) so that fragments amenable to DD compression can be scheduled accordingly.

7.3 Tensor networks and matrix product state (MPS) simulation

Tensor-network methods and matrix-product-state representations are the method of choice for low-entanglement, structured circuits (and for many condensed-matter inspired problems). Reviews and algorithmic foundations for MPS/TN algorithms are widely available and establish the key dependence of cost on bond dimension (χ) and contraction order [17]. QuASAr uses MPS/TN cost formulas as a primary basis for bond-dimension estimation and for deciding when to prefer tensor-network backends.

7.4 Stabilizer-rank, magic complexity and hybrid classical algorithms

For circuits that mix Clifford and non-Clifford gates, results on stabilizer-rank / magic-state complexity (Bravyi and coauthors and follow-up works) provide a theoretical explanation for why tableau methods break down as the number of non-Clifford gates grows [6]. These works also inspired techniques for hybrid simulation (e.g., stabilizer decompositions) that trade runtime against the number of T gates. QuASAr uses stabilizer-count histograms in its micro-estimators and models stabilizer-rank growth in its tableau cost functions.

7.5 Hybrid and convertible representations

A growing body of work explores representations that bridge the gaps between DDs, tableau/stabilizer reasoning, and tensor networks. LIMDD (Local Invertible Map Decision Diagrams) and related recent proposals (e.g., LimTDD) explicitly aim to unify stabilizer-like compressibility with tensor-style locality, enabling more compact intermediate forms and conversions between representations [15, 22]. Such hybrid structures motivate QuASAr's SSD and bridge-tensor concepts: rather than fully materializing a state vector, QuASAr extracts small, canonical interfaces that are often sufficient to transition between backends at low cost.

7.6 Multi-backend simulators and adaptive execution

General-purpose simulator frameworks (Qiskit Aer, TensorNetwork-based simulators, hybrid Schrödinger–Feynman approaches) typically provide multiple static backends or user-configurable options (state-vector, stabilizer, MPS, sampling / Feynman path) but do not perform fine-grained, runtime backend switching driven by a cost-based optimizer [2, 16, 23]. Recent work on "simulation paths" for decision diagrams and on hybrid contraction strategies for tensor networks highlights the importance of planning the sequence of operations and the representation path through the circuit; this is

close in spirit to QuASAr's plan enumeration and switching-aware DP [7, 8].

7.7 Conversion, approximation, and bounded errors

Approaches that trade accuracy for compressibility (approximate DDs, truncated tensor networks, low-rank approximations) provide useful primitives for conversion and staged materialization. Approximating decision-diagrams and tensor truncation strategies have been proposed to reduce resource requirements while controlling error [14, 24]. QuASAr's staged conversion primitives (boundary extraction, local-window conversion, staged MPS approximations) are designed to exploit such approximations conservatively and to track the induced error so that simulation fidelity can be controlled by policy (e.g., strict correctness vs. approximate speedup).

7.8 How QuASAr differs

While many prior systems and data structures provide pieces of the functionality QuASAr requires (fast tableau simulation, production DD implementations, tensor-network toolkits, and hybrid DD/TN proposals), QuASAr's novelty is to combine: (1) *theory-grounded* cost models for multiple backends, (2) a compact interface descriptor (SSD) that enables low-cost partial conversions, and (3) a database-inspired, switching-aware optimizer that performs DP-based plan enumeration and runtime re-optimization. This combination of precise cost accounting, partial-conversion primitives, and adaptive plan search is — to our knowledge — absent from existing simulation frameworks and directly targets the kinds of heterogeneous circuits encountered in quantum-data-management use cases.

8 Experimental Evaluation

This section provides a structured skeleton for the experimental evaluation of QuASAr. It is intentionally modular: each experiment entry contains the objective, methodology, inputs, metrics, and placeholders for figures/tables so that numerical results and plots can be dropped in later without restructuring. The experiments are designed to answer the evaluation questions that follow and to be reproducible from the project benchmark harness.

8.1 Evaluation goals and research questions

We evaluate QuASAr with the following primary goals and research questions (RQs):

RQ1 (Effectiveness). How much wall-clock time and memory does QuASAr save relative to single-backend baselines (state-vector, DD, MPS, tableau) across representative workloads?

RQ2 (Switching cost vs. benefit). How often does QuASAr choose to switch backends, and when it does, does the benefit outweigh the conversion penalty?

RQ3 (Optimizer quality and overhead). What is the quality of the produced plans (compared to an oracle / exhaustive search on small circuits) and what is the CPU overhead of plan enumeration and re-optimization?

RQ4 (Robustness & adaptivity). How robust is QuASAr under estimation errors and adversarial circuits (e.g., sudden entanglement spikes)? How quickly and effectively does the adaptive re-optimizer correct poor initial plans?

RQ5 (Micro-performance). What are the microbenchmarks for conversion primitives (SSD extraction, boundary conversion, local-window extraction, stabilizer learning) and how do their measured times match the calibrated cost model?

8.2 High-level methodology

- For each experiment we describe the benchmark circuits, baseline configurations, measurement protocol (repetitions, warm-up), and the metrics to be collected.
- All timings are wall-clock times measured with high-resolution timers. Memory is peak resident set size measured via OS tools (e.g., `psutil` or `/usr/bin/time -v`). We record the number of backend switches, conversion times, and plan-cache hit rates.
- Each experiment is run ≥ 5 independent times (random seeds where applicable); we report mean and standard deviation. For plots we include error bars (95% confidence intervals) when applicable.
- All experiments are reproducible via the provided benchmark harness; we document exact command-lines and environment variables in the appendix.

8.3 Setup: hardware, software and baseline configurations

Hardware. Describe the machines used for evaluation (example template below; fill with actual values):

- **Server A:** CPU model, number of cores/threads, RAM, OS, BLAS/LAPACK implementation (e.g., MKL/OpenBLAS), Python version.
- **Laptop B:** CPU model, cores, RAM, OS, Python version (useful for low-resource experiments).

Software. Record key versions:

- QuASAr (git commit hash)
- Stim version
- MQT Core / DD package version
- Python, pybind11, compiler version (gcc/clang) and CMake version

Baselines and configurations. We compare QuASAr against the following single-backend baselines:

- **SV:** State-vector (single-threaded / multi-threaded variants).
- **DD:** Decision-diagram simulation (MQT Core) with default variable ordering.
- **MPS:** MPS/tensor-network simulator with truncation tolerance X (document here).
- **TAB:** Stim tableau simulator for Clifford-dominant circuits.

For fairness, each baseline uses the same underlying numerical libraries where applicable (e.g., same BLAS).

8.4 Workloads / benchmark suite

We use a curated set of circuits that exercise different compression and entanglement regimes. For each circuit we will provide (in the final paper) the circuit source (QASM / Quil / proprietary generator), qubit count, gate count, and a short characterization.

8.5 Metrics

We collect the following metrics for each run:

Table 3: Benchmark suite (placeholder — fill with exact circuits used).

Short name	Qubits	Gates	Description / reason for inclusion
Clifford-EC	100	10k	Error-correction syndrome circuits (tableau-friendly)
Ripple-Add	40	5k	Arithmetic with regular structure (DD-friendly)
VQE-Chain	50	200	Low-depth nearest-neighbor ansatz (MPS-friendly)
Random-Hybrid	30	200	Mixed Clifford/T gates, stresses switching logic
Recur-Subroutine	20	2k	Repeated subroutines to test plan caching

- **Total runtime** (wall clock)
- **Peak memory** (RSS)
- **Number of backend switches**
- **Total conversion time** (sum of conversion primitives)
- **Plan generation time** (optimizer overhead)
- **Plan cache hits & re-optimization count**
- **State fidelity / correctness**: end-to-end numerical comparison against a reference state (where feasible), e.g., fidelity $F = \langle\psi_{\text{ref}}|\psi_{\text{quasar}}\rangle$.

8.6 Experiment suite (skeleton)

Below we define the specific experiments. For each experiment we provide a recommended set of figures and tables (placeholders) to be filled with results.

E1 — Microbenchmarks and calibration. **Objective:** Measure primitive operation costs used to calibrate the switching-penalty model: SVD times for different bond-dimensions χ, DD frontier extraction times vs. frontier-size, stabilizer learning time vs. window size, and ingestion times for target backends.

Methodology: For each primitive, run across a parameter sweep (e.g., $\chi \in \{8, 16, 32, 64, 128\}$, frontier size $r \in \{10, 50, 200, 1000\}$, window sizes $w = 2..8$) and record time / memory.

Placeholders:

- Figure: SVD runtime vs. χ (log-log plot)
- Figure: DD frontier extraction time vs. frontier size
- Table: Stabilizer-learning time vs. window size (mean ± std)

E2 — End-to-end comparison (effectiveness). **Objective:** Compare QuASAr against single-backend baselines on the full benchmark suite. Answer RQ1.

Methodology: For each circuit in Table 3, run QuASAr and each baseline, record metrics. Repeat runs to obtain variance.

Placeholders:

- Table: Runtime / Memory summary (rows = circuits, columns = QuASAr + baselines)
- Bar chart: Relative speedup of QuASAr over best baseline
- Table: Number of backend switches and total conversion time per circuit

E3 — Switching cost vs. benefit (ablation). **Objective:** Quantify when switching helps and the sensitivity to conversion penalties. Evaluate decision boundaries where QuASAr prefers to switch.

Methodology: Run QuASAr under varied artificial conversion-penalty multipliers $\alpha \in \{0.5, 1, 2, 5\}$ and observe plan changes and end-to-end performance.

Placeholders:

- Heatmap: Plan choice (backend per fragment) as a function of α
- Line plots: End-to-end runtime vs. α

E4 — Planner quality and overhead. **Objective:** Measure the optimizer's plan quality and its runtime overhead. Compare to an *oracle* plan on small circuits (exhaustive search) and to greedy heuristics.

Methodology: On small circuits (e.g., up to 20 qubits and limited gate count), compute:

- Exhaustive optimal plan (oracle)
- QuASAr's DP plan
- Greedy plan (local best backend per fragment)

Measure the plan cost and planner runtime.

Placeholders:

- Table: Plan cost and planner time for each method (oracle / DP / greedy)
- CDF: Distribution of plan-cost gaps (DP vs. oracle)

E5 — Robustness and adaptivity. **Objective:** Evaluate adaptive re-optimization under estimation drift and adversarial circuits.

Methodology:

- Introduce perturbations to the circuit during execution (e.g., inject extra entangling gates unexpectedly) or artificially corrupt initial cost estimates.
- Measure time-to-recover (time until planner corrects plan) and penalty incurred during recovery.

Placeholders:

- Timeline plot: Execution progress showing when re-optimization triggers occur and subsequent performance recovery.
- Table: Recovery time and overhead for different perturbation magnitudes.

E6 — Plan cache effectiveness and repeated workloads. **Objective:** Measure benefits of plan caching for repeated or parameterized circuits.

Methodology: Evaluate repeated invocations of parameterized circuits (e.g., VQE with varying parameters; repeated subroutine calls) and measure plan-cache hit rate, speedup, and conversion reuse.

Placeholders:

- Table: Plan cache hit rate vs. number of repetitions
- Line plot: Cumulative speedup across repeated runs (cold vs. warm cache)

8.7 Statistical treatment and significance

When comparing methods we apply standard statistical tests:

- Use paired t-tests or Wilcoxon signed-rank tests (non-parametric) to assess significance in runtime differences across repeated runs.
- For multiple comparisons across many circuits, apply a Bonferroni correction or report false-discovery rate controls.
- Report effect sizes (Cohen's d) alongside p-values.

Table 4: (Placeholder) Example runtime / memory summary format. Fill with measured values (mean ± std).

Circuit	QuASAr time (s)	Best baseline time (s)	QuASAr peak mem (GB)	Baseline peak mem (GB)
Clifford-EC	–	–	–	–
Ripple-Add	–	–	–	–
VQE-Chain	–	–	–	–

8.8 Figures and tables (placeholders)

Below are LaTeX placeholders for common figures/tables used across experiments. Replace the <FIGURE/TABLE> markers with actual images or tabular data.

Figure 2: (Placeholder) End-to-end runtime comparison across benchmarks.

8.9 Execution recipes and reproducibility notes

We recommend the following concrete invocation pattern (to be adapted for your harness):

```
# Example experiment run (placeholder)
python3 run_benchmark.py --circuit <circuit-file> --mode quasar \
    --quasar-config config/quasar.yml \
    --reps 10 --out results/<circuit>-quasar.json

python3 run_benchmark.py --circuit <circuit-file> --mode baseline_dd \
    --reps 10 --out results/<circuit>-dd.json
```

Document the exact command-lines used for each reported number in the appendix so reviewers (and readers) can reproduce the results.

9 Conclusion and Future Work

This paper introduced QuASAr, a declarative architecture for adaptive quantum circuit simulation that automatically optimizes method selection and resource utilization. Our key contributions include:

- The first declarative API for quantum circuit simulation that abstracts method selection complexity from users
- A novel two-level optimization strategy combining method-based partitioning with parallelism optimization
- Comprehensive evaluation demonstrating significant performance improvements over manual approaches
- A unified framework supporting multiple simulation backends with intelligent coordination

QuASAr achieves up to 4.1× speedup over manual method selection while maintaining high accuracy and providing a simple, declarative interface. The system successfully handles diverse quantum circuits and automatically adapts to different computational environments.

9.1 Future Directions

Several directions for future work emerge from this research:

Machine Learning Enhancement: Incorporating more sophisticated machine learning models for method selection, potentially using reinforcement learning to adapt to user preferences and system characteristics over time.

Distributed Execution: Extending QuASAr to distributed computing environments, enabling simulation of larger circuits across multiple machines with intelligent load balancing and communication optimization.

Approximate Simulation: Integrating approximate simulation methods that trade accuracy for performance, with automatic error bound management and adaptive precision control.

Hardware Integration: Adapting QuASAr for hybrid classical-quantum simulation, where parts of circuits execute on quantum hardware while others use classical simulation, requiring sophisticated resource allocation and scheduling.

Domain-Specific Optimization: Developing specialized optimizations for specific quantum algorithm families, such as optimization algorithms, quantum machine learning, and quantum chemistry simulations.

Real-Time Adaptation: Implementing real-time performance monitoring and adaptation capabilities that can adjust simulation strategies during execution based on observed performance characteristics.

Energy Efficiency: Incorporating energy consumption models into the optimization framework, enabling trade-offs between performance and energy usage for sustainable quantum simulation.

The success of QuASAr demonstrates the value of declarative approaches in quantum computing systems. As quantum circuits become larger and more complex, automated optimization will become increasingly critical for practical quantum computing applications. The principles and techniques developed in QuASAr provide a foundation for future advances in quantum simulation and broader quantum computing system design.

References

[1] 2024. Cirq import/export circuits. https://quantumai.google/cirq/build/interop.

[2] 2024. Qiskit documentation: QuantumCircuit class. https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.QuantumCircuit.

[3] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Physical Review A* 70, 5 (2004), 052328. doi:10.1103/PhysRevA.70.052328

[4] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: continuously adaptive query processing. *SIGMOD Rec.* 29, 2 (May 2000), 261–272. doi:10.1145/335191.335420

[5] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J Bremner, John M Martinis, and Hartmut Neven. 2018. Characterizing Quantum Supremacy in Near-Term Devices. *Nature Physics* 14, 6 (2018), 595–600.

[6] Sergey Bravyi and David Gosset. 2016. Improved Classical Simulation of Quantum Circuits Dominated by Clifford Gates. *Physical Review Letters* 116 (Jun 2016), 250501. Issue 25.

[7] Lukas Burgholzer, Hartwig Bauer, and Robert Wille. 2021. Hybrid Schrödinger-Feynman Simulation of Quantum Circuits With Decision Diagrams . In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE Computer Society, Los Alamitos, CA, USA, 199–206. doi:10.1109/QCE52317.2021.00037

[8] Lukas Burgholzer, Alexander Ploier, and Robert Wille. 2023. Simulation Paths for Quantum Circuit Simulation With Decision Diagrams What to Learn From Tensor Networks, and What Not. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 42, 4 (April 2023), 1113–1122. doi:10.1109/TCAD.2022.3197969

[9] Lukas Burgholzer, Yannick Stade, Tom Peham, and Robert Wille. 2025. MQT Core: The Backbone of the Munich Quantum Toolkit (MQT). *Journal of Open Source Software* 10, 108 (2025), 7478. doi:10.21105/joss.07478

[10] NVIDIA cuQuantum Team. 2024. NVIDIA cuQuantum. https://docs.nvidia.com/cuda/cuquantum/22.07.1/cutensornet/overview.html. Accessed: 2024-08-14.

[11] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (2021), 497. doi:10.22331/q-2021-07-06-497

[12] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv.* 25, 2 (June 1993), 73–169. doi:10.1145/152610.152611

[13] Rihan Hai, Shih-Han Hung, Tim Coopmans, Tim Littau, and Floris Geerts. 2025. Quantum Data Management in the NISQ Era: Extended Version. https://arxiv.org/abs/2409.14111

[14] Stefan Hillmich, Alwin Zulehner, Richard Kueng, Igor L. Markov, and Robert Wille. 2022. Approximating Decision Diagrams for Quantum Circuit Simulation.

ACM Transactions on Quantum Computing 3, 4, Article 22 (July 2022), 21 pages. doi:10.1145/3530776

[15] Xin Hong, Aochu Dai, Dingchao Gao, Sanjiang Li, Zhengfeng Ji, and Mingsheng Ying. 2025. LimTDD: A Compact Decision Diagram Integrating Tensor and Local Invertible Map Representations. arXiv:2504.01168 [cs.DS] https://arxiv.org/abs/2504.01168

[16] Salvatore Mandrà, Jeffrey Marshall, Eleanor G. Rieffel, and Rupak Biswas. 2021. HybridQ: A Hybrid Simulator for Quantum Circuits. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*. 99–109. doi:10.1109/QCS54837.2021.00015

[17] Román Orús. 2014. A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States. *Annals of Physics* 349 (2014), 117–158.

[18] John Preskill. 2018. Quantum Computing in the NISQ Era and Beyond. *Quantum* 2 (Aug. 2018), 79.

[19] Kartikey Sarode. 2024. Circuit Partitioning and Full Circuit Execution: A Comparative Study of GPU - Based Quantum Circuit Simulation . In *2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 177–187. doi:10.1109/HiPC62374.2024.00026

[20] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. doi:10.1145/582095.582099

[21] Guifré Vidal. 2003. Efficient Classical Simulation of Slightly Entangled Quantum Computations. *Physical Review Letters* 91, 14 (2003), 147902.

[22] Lieuwe Vinkhuijzen, Tim Coopmans, David Elkouss, Vedran Dunjko, and Alfons Laarman. 2023. LIMDD: A Decision Diagram for Simulation of Quantum Computing including Stabilizer States. *Quantum* 7 (2023), 1108.

[23] Xiao Yuan, Jinzhao Sun, Junyu Liu, Qi Zhao, and You Zhou. 2021. Quantum Simulation with Hybrid Tensor Networks. *Phys. Rev. Lett.* 127 (Jul 2021), 040501. Issue 4. doi:10.1103/PhysRevLett.127.040501

[24] Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to Efficiently Handle Complex Values? Implementing Decision Diagrams for Quantum Computing. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–7. doi:10.1109/iccad45719.2019.8942057

[25] Alwin Zulehner and Robert Wille. 2018. Advanced Simulation of Quantum Computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 38, 5 (2018), 848–859.