A.R.B.O.R.

Advanced Reasoning By Ontological Rules

Enterprise Technical Documentation

The Contextual Discovery Engine

Version 1.0

February 7, 2026

Curated AI Discovery for Any Domain

Confidential — Internal Use Only

Contents

List of Figures

List of Tables

Part I

Executive Overview

Chapter 1

Introduction

In an era defined by information abundance, the fundamental challenge facing individuals and organizations has shifted dramatically. The problem is no longer the scarcity of options, but rather the overwhelming proliferation of choices that paralyzes decision-making. Whether searching for a bespoke tailor in Milan, a luxury property in Paris, or a specialized consultant in London, users find themselves drowning in a sea of search results that fail to capture the nuanced qualities that truly matter.

Traditional search engines excel at retrieval by keyword matching, but they fundamentally misunderstand the nature of discovery. When someone searches for "a place with Neapolitan tailoring and an intimate atmosphere," they are not looking for a list of stores that happen to contain those words in their descriptions. They are seeking an experience—a synthesis of craftsmanship, ambiance, heritage, and that ineffable quality the Italians call *sprezzatura*.

A.R.B.O.R.—**Advanced Reasoning By Ontological Rules**—represents a paradigm shift in how we approach the discovery problem. Rather than treating search as a mechanical matching exercise, ARBOR conceptualizes it as a curatorial act, one that requires understanding, context, and taste.

1.1 The Discovery Problem

The modern discovery landscape is characterized by several interconnected challenges that conventional approaches have failed to adequately address.

1.1.1 The Paradox of Choice

Psychologist Barry Schwartz famously articulated the paradox of choice: as the number of options increases, so does the anxiety associated with making decisions, while satisfaction with the chosen option often decreases. In the context of curated experiences—

whether selecting a restaurant for an anniversary, finding a craftsman for bespoke shoes, or identifying the perfect property—this paradox manifests acutely.

Consider the experience of planning a special dinner in a major metropolitan area. A simple Google search returns thousands of results. Review aggregators like TripAdvisor or Yelp provide ratings, but these suffer from well-documented biases: the vocal minority effect, astroturfing, the regression to mediocrity in scoring. The reviews themselves are unstructured text that requires significant cognitive effort to parse, and the reader has no way to calibrate whether a particular reviewer shares their values or sensibilities.

1.1.2 The Limitations of Keyword Search

Traditional search operates on the principle of lexical matching enhanced by various ranking signals. While remarkably effective for informational queries ("what is the capital of France?") or navigational queries ("OpenAI website"), this paradigm breaks down when the user's intent is exploratory or when the relevant attributes are subjective and multidimensional.

A user searching for "quiet restaurant with excellent wine list" expresses two explicit criteria, but implicitly communicates a desire for sophistication, attention to detail, and perhaps a certain price range. Even more challenging are queries that express lifestyle alignment: "somewhere my design-conscious friends would appreciate" or "the kind of place a *Financial Times* reader would frequent." These queries require a form of cultural fluency that keyword matching cannot provide.

1.1.3 The Fragmentation of Data

Information about entities of interest—restaurants, boutiques, properties, professionals—is scattered across numerous platforms, each capturing only partial facets of the whole. Google Maps provides location and basic operational details. Instagram reveals visual aesthetics. Reviews on specialized platforms offer subjective assessments. The entity's own website presents a curated self-image. Industry awards and press coverage contribute additional signals.

No existing system synthesizes these diverse data sources into a coherent, queryable representation of an entity's true character. Users must manually aggregate and interpret information from multiple sources, a process that is both time-consuming and prone to error.

1.2 The ARBOR Vision

ARBOR addresses these challenges through a fundamentally different approach to discovery. Rather than attempting to match queries to documents, ARBOR maintains a rich, multidimensional representation of entities and employs AI agents to understand user intent, navigate this knowledge space, and synthesize recommendations that match not just explicit criteria but implicit preferences and contextual factors.

1.2.1 Curated AI Discovery

The term "curated" is deliberately chosen. Traditional curation—whether in museums, magazines, or retail—involves a human expert making selections based on deep domain knowledge, aesthetic sensibility, and understanding of the audience. ARBOR replicates this curatorial function through a combination of:

1. **Structured Knowledge Representation**: Entities are characterized along multiple dimensions relevant to the domain, captured in what we term the "Vibe DNA"—a vector of scores that encodes subjective qualities like atmosphere, craftsmanship, exclusivity, and service.

2. **Relationship Graphs**: Entities exist not in isolation but in a web of relationships—a tailor trained by a master, a restaurant featuring wines from a particular producer, a property designed by a noted architect. These connections enrich discovery by enabling traversal along meaningful dimensions.

3. **Intelligent Interpretation**: Natural language queries are parsed by agents that understand domain-specific vocabulary, cultural context, and the implicit signals within requests.

4. **Synthesis, Not Just Retrieval**: Rather than returning a ranked list, ARBOR's Curator agent synthesizes a response that explains why particular entities match the user's needs, drawing on the structured knowledge to provide substantive justification.

1.2.2 Domain Agnosticism

A central architectural principle of ARBOR is domain agnosticism. While the initial implementation focuses on lifestyle and luxury retail, the system is designed to adapt to any domain through configuration rather than code modification. The domain configuration file specifies:

- The entity types and their attributes
- The dimensions of the Vibe DNA and their weights
- The relationship types in the knowledge graph
- The Curator persona and vocabulary
- Example queries for training and evaluation

This design enables rapid deployment to new verticals—real estate, hospitality, professional services, recruiting—without engineering effort beyond data ingestion and configuration.

1.3 Document Overview

This technical documentation provides a comprehensive reference for all aspects of the ARBOR system. It is organized into seven parts, each addressing a distinct aspect of the platform.

Part I: Executive Overview (the present section) introduces the problem domain, articulates the vision, and positions ARBOR relative to alternative approaches. It is intended for technical and non-technical readers alike who seek to understand what ARBOR does and why it matters.

Part II: System Architecture presents the high-level architecture, data flows, and the three-database "Knowledge Trinity" that underlies ARBOR's capabilities. This section provides the conceptual foundation necessary to understand subsequent technical details.

Part III: Core Modules offers deep technical documentation of the backend systems: the ingestion pipeline, the discovery engine, the machine learning components, and the event-driven architecture. This section is intended for developers who will extend or maintain the system.

Part IV: Frontend & User Experience documents the web and mobile interfaces, the component library, and the administrative dashboard used by curators and operators.

Part V: Infrastructure & Operations covers deployment, observability, and security—the operational concerns necessary to run ARBOR at scale with reliability and safety.

Part VI: Testing & Quality Assurance details the testing strategy, from unit tests through chaos engineering, explaining how the system maintains quality across its many components.

Part VII: Appendices provides reference material including the complete API specification, configuration schema documentation, and a glossary of terms.

Each chapter is designed to be self-contained while cross-referencing related content in other sections. Code examples are provided where they illuminate concepts, but the documentation prioritizes explanation over exhaustive code listings—the latter are available in the source repository.

Chapter 2

Objectives and Vision

The development of ARBOR is guided by a coherent set of objectives that span technical, user experience, and business dimensions. These objectives inform every architectural decision and feature prioritization, ensuring that the system delivers genuine value rather than merely technical sophistication.

2.1 Strategic Vision

Mission Statement

To democratize access to expertly curated recommendations by combining artificial intelligence with structured domain knowledge, enabling anyone to discover options that match their preferences with the precision and insight of a personal concierge.

This mission reflects a belief that quality curation should not be the exclusive province of the wealthy or well-connected. A first-time visitor to a city deserves access to the same caliber of recommendations that a longtime resident with extensive networks might possess. A small business owner seeking professional services should be able to find the right fit as efficiently as a corporation with dedicated procurement teams.

The vision extends beyond individual users to encompass the entities themselves. For a small atelier producing exceptional work, ARBOR offers the possibility of discovery by precisely the clientele who would appreciate their craft—without requiring the marketing budgets of larger competitors.

2.2 Technical Objectives

The technical objectives translate the strategic vision into concrete engineering goals.

2.2.1 Performance Requirements

The system must deliver responses with latency appropriate for conversational interaction. Our target is end-to-end response time under 2.5 seconds at the 95th percentile, measured from query submission to complete response delivery. This constraint requires careful optimization across the entire pipeline: query parsing, agent orchestration, database queries, LLM inference, and response synthesis.

Achieving this latency target while maintaining response quality necessitates a multi-faceted approach:

- **Semantic Caching**: Similar queries can be served from cached responses, dramatically reducing latency and LLM costs for common patterns.
- **Parallel Agent Execution**: Independent agents execute concurrently rather than sequentially.
- **Optimized Vector Search**: Hybrid search combining sparse and dense representations enables sub-100ms retrieval even at scale.
- **Cost-Aware Model Routing**: Simple queries route to faster, less expensive models; complex queries engage more capable systems.

2.2.2 Scalability Targets

ARBOR is architected to scale from initial deployment to enterprise production without fundamental redesign. The system targets:

- **Entity Capacity**: Support for 1 million entities per domain without performance degradation.
- **Query Throughput**: 100 concurrent users with consistent sub-3-second response times.
- **Horizontal Scaling**: Stateless API servers enable linear scaling via additional instances.
- **Database Federation**: Each database in the Knowledge Trinity scales independently according to its workload characteristics.

2.2.3 Reliability Standards

For production deployment, ARBOR targets 99.9% availability measured on a monthly basis, permitting approximately 43 minutes of downtime per month. This is achieved through:

- Redundant deployments across availability zones
- Graceful degradation when individual components fail
- Circuit breakers preventing cascade failures
- Comprehensive health checks and automated recovery

2.3 User Experience Objectives

Technical excellence means nothing if the user experience fails to deliver value. ARBOR's UX objectives prioritize genuine utility over feature proliferation.

2.3.1 Conversational Fluency

The primary interaction mode is natural language conversation. Users should be able to express their needs in their own words without learning query syntax or filtering vocabularies. The system must handle:

- Vague initial queries that require clarification
- Multi-turn conversations that refine preferences iteratively
- Domain-specific vocabulary and colloquialisms
- Implicit preferences inferred from context
- Follow-up questions referencing previous recommendations

2.3.2 Explanation, Not Just Recommendation

Unlike opaque recommendation systems that present options without justification, ARBOR's Curator provides explanatory context for each suggestion. When recommending a restaurant, the system might note: "This establishment shares the same commitment to seafood provenance as the venue you mentioned enjoying, but offers a more intimate atmosphere that sounds better suited to your anniversary celebration."

This explanatory approach serves multiple purposes:

1. **Trust Building**: Users can evaluate whether the reasoning aligns with their actual preferences.
2. **Preference Refinement**: Explanations help users articulate aspects they hadn't consciously considered.

3. **Knowledge Transfer**: Over time, users develop their own curatorial vocabulary and frameworks.

2.3.3 Graceful Handling of Limitations

The system must honestly acknowledge the boundaries of its knowledge. When data is incomplete, when preferences conflict with available options, or when a query falls outside the system's competence, the response should be transparent rather than confabulated.

2.4 Business Objectives

While ARBOR's technical and user experience objectives serve intrinsic quality goals, the system must also support sustainable business models.

2.4.1 Multi-Tenancy and White-Labeling

The architecture supports deployment as a multi-tenant platform or as white-labeled instances for individual clients. A luxury concierge service might deploy ARBOR with their branding, populated with entities they have personally vetted. A city tourism authority might offer a discovery interface for visitors. A real estate platform might integrate ARBOR for property search.

2.4.2 Monetization Flexibility

The platform's modular design supports various commercialization models:

- **Subscription Access**: Tiered plans with rate limits and feature differentiation.
- **API Licensing**: B2B access for integration into third-party applications.
- **Featured Placement**: Entities can pay for enhanced visibility without compromising recommendation quality.
- **Analytics Services**: Aggregated insights about user preferences and market trends.

2.4.3 Operational Efficiency

The system minimizes manual intervention required for ongoing operation:

- **Automated Ingestion**: New entities are continuously discovered and ingested without curator involvement.
- **Quality Maintenance**: Drift detection and automated validation flag entities requiring review.
- **Cost Optimization**: Intelligent caching and model routing minimize inference costs.
- **Self-Healing**: Automated recovery from common failure modes reduces operator burden.

2.5 Success Metrics

Achievement of these objectives is measured through a comprehensive metrics framework organized into four categories.

2.5.1 Technical Health Metrics

Metric	**Target**	**Critical Threshold**
Response Latency P50	< 1.5s	< 2.0s
Response Latency P95	< 2.5s	< 3.5s
Response Latency P99	< 4.0s	< 6.0s
API Availability	$> 99.9\%$	$> 99.5\%$
Error Rate	$< 0.1\%$	$< 1.0\%$
Cache Hit Rate	$> 40\%$	$> 25\%$

Table 2.1: Technical performance targets

2.5.2 Quality Metrics

User satisfaction and recommendation quality are assessed through:

- **Recommendation Click-Through Rate**: Proportion of recommendations that users pursue.
- **Conversion Rate**: Recommendations that result in actual visits or transactions.
- **Return User Rate**: Users who return for additional queries.
- **Session Depth**: Average number of queries per user session.

- **Curator Accuracy**: Expert assessment of recommendation appropriateness.

2.5.3 Operational Metrics

- **Ingestion Pipeline Throughput**: Entities processed per hour.
- **Data Freshness**: Age of entity information relative to sources.
- **LLM Cost per Query**: Average inference cost per user interaction.
- **Infrastructure Cost per Active User**: Total operational cost normalized by usage.

2.5.4 Business Metrics

- **Monthly Active Users**: Distinct users with at least one query.
- **User Acquisition Cost**: Marketing spend per new user.
- **Revenue per User**: Monetization effectiveness.
- **Entity Coverage**: Proportion of relevant entities in the domain captured.

2.6 Non-Objectives

Clarity about what ARBOR is *not* trying to achieve is equally important for guiding development decisions.

2.6.1 General-Purpose Search

ARBOR is not a replacement for Google or other general search engines. It is purpose-built for curated discovery within specific domains. Queries about general knowledge, current events, or topics outside configured domains are explicitly out of scope.

2.6.2 Real-Time Booking or Transactions

While ARBOR may link to booking platforms, it does not itself handle reservations, purchases, or other transactions. The focus remains on discovery and recommendation; transactional complexity is delegated to specialized systems.

2.6.3 User-Generated Content Platform

ARBOR is not a review site or social platform. While it ingests and analyzes reviews from external sources, it does not solicit or host user-contributed content. The knowledge base is maintained through automated ingestion and curator validation, not crowdsourcing.

2.6.4 Complete Automation of Curation

Despite significant AI capabilities, ARBOR is designed to augment rather than replace human curators. The system surfaces candidates and provides analysis, but humans retain final authority over entity validation, relationship creation, and quality standards. Full automation would sacrifice the discernment that makes curation valuable.

Chapter 3

Competitive Analysis

Understanding ARBOR's position in the broader landscape of discovery and recommendation systems illuminates both its unique value and the limitations of existing approaches. This analysis examines the primary categories of alternatives, their strengths, and the structural gaps that ARBOR addresses.

3.1 Traditional Search Engines

Google, Bing, and other general search engines represent the default tool for most discovery tasks. Their strengths are undeniable: comprehensive indexing, sophisticated ranking algorithms refined over decades, and near-universal availability.

3.1.1 Strengths

Search engines excel at breadth of coverage and speed of retrieval. For any given query, Google will return results in milliseconds, drawing on an index of billions of pages. The PageRank algorithm and its successors effectively identify authoritative sources, and continuous crawling ensures reasonable freshness for actively maintained content.

For navigational queries—finding a known entity's website—search engines are nearly optimal. For informational queries with factual answers, featured snippets and knowledge panels provide direct responses without requiring users to visit external pages.

3.1.2 Structural Limitations

The fundamental limitation of search engines for curated discovery lies in their document-centric paradigm. Search engines retrieve and rank documents; they do not understand entities or their attributes. When a user searches for “intimate restaurant with natural wine,” the search engine matches those keywords against web pages, not against a structured representation of restaurants and their characteristics.

This leads to several specific deficiencies:

- **No Attribute Comparison**: Search engines cannot compare entities along specific dimensions. “Which of these restaurants has a better wine list?” is unanswerable.
- **No Relationship Navigation**: Connections between entities—a chef trained at a notable establishment, a sommelier previously at a celebrated wine bar—are invisible to keyword search.
- **SEO Distortion**: Results are biased toward entities with sophisticated search engine optimization, which correlates poorly with intrinsic quality.
- **Review Aggregation Failure**: While Google indexes reviews, it cannot synthesize insights across review corpora or weight reviewer credibility.
- **Contextual Blindness**: The same query yields the same results regardless of user context, occasion, or implicit preferences.

3.2 Review Aggregators

Platforms like TripAdvisor, Yelp, and Google Maps reviews provide structured ratings and user-generated reviews for entities, particularly in hospitality and dining.

3.2.1 Strengths

Review aggregators democratize access to opinions, enabling anyone to benefit from collective experience. For broadly popular establishments, the volume of reviews provides statistical reliability. The platforms offer convenient filtering by category, location, and rating range.

3.2.2 Structural Limitations

Review aggregators suffer from well-documented biases that systematically distort their utility for discerning users:

- **Self-Selection Bias**: Reviewers are not representative of all customers. They skew toward those with extreme experiences (very positive or very negative) and those with time and inclination to write reviews.

- **Astroturfing**: Fake reviews—both positive (by the entity or its promoters) and negative (by competitors)—are endemic despite platform efforts at detection.
- **Temporal Decay**: Reviews from years ago affect current ratings even when establishments have changed ownership, chefs, or quality.
- **Rating Inflation**: Social pressure and platform mechanics push ratings toward the high end, compressing meaningful differentiation into a narrow band between 4.0 and 5.0 stars.
- **Reviewer Mismatch**: A user cannot assess whether a reviewer shares their values. A glowing review from someone who prioritizes portion size conveys little to someone who prioritizes ingredient quality.
- **Attribute Conflation**: A single rating collapses multiple dimensions (food, service, atmosphere, value) into one number, obscuring important distinctions.

ARBOR addresses these limitations by applying AI analysis to extract signal from review noise, by maintaining multi-dimensional attribute scoring, and by providing curatorial context that helps users interpret recommendations.

3.3 Domain-Specific Guides

Michelin, Zagat, Gambero Rosso, and similar curated guides offer expert-vetted selections and ratings. These represent the traditional model of human curation that ARBOR seeks to augment with technology.

3.3.1 Strengths

Expert curation provides consistent standards applied by trained assessors. The brand reputation of established guides confers trust. The selectivity of inclusion itself signals quality—absence from the Michelin guide says something meaningful.

These guides also maintain historical context and recognize trajectory: an establishment may be noted as “rising” or “declining” based on longitudinal assessment.

3.3.2 Structural Limitations

- **Coverage Constraints**: Human curation does not scale. Michelin covers only major cities, and even within those cities, only a fraction of establishments. Countless excellent venues remain unexamined.

- **Update Frequency**: Print publication cycles and assessment logistics mean information can be months or years out of date.

- **Categorical Focus**: Guides typically specialize in a single domain (restaurants, hotels). Users seeking cross-category recommendations find no integrated solution.

- **No Personalization**: A Michelin star means the same thing regardless of whether the user prefers formal or casual, adventurous or traditional. Personal preferences cannot modulate the guide's assessments.

- **Query Inflexibility**: Guides are structured for browsing by location and price, not for answering natural language queries about specific combinations of attributes.

3.4 Recommendation Systems

Netflix, Spotify, Amazon, and other platforms deploy sophisticated recommendation algorithms to surface relevant content and products.

3.4.1 Strengths

Modern recommendation systems leverage vast behavioral data to predict user preferences with remarkable accuracy. Collaborative filtering identifies users with similar tastes; content-based filtering matches items to demonstrated preferences; hybrid approaches combine multiple signals.

These systems operate at scale, personalizing experiences for millions of users simultaneously. They continuously learn from user interactions, improving recommendations over time.

3.4.2 Structural Limitations

Despite their sophistication, these systems exhibit limitations relevant to curated discovery:

- **Cold Start Problem**: New users and new items lack the interaction history needed for effective recommendations. A first-time visitor cannot benefit from behavioral patterns.

- **Filter Bubbles**: Optimization for engagement can trap users in increasingly narrow preference spaces, reducing serendipitous discovery.

- **Black Box Opacity**: Users cannot understand why particular recommendations appear, undermining trust and preventing meaningful feedback.
- **Behavioral vs. Stated Preferences**: These systems optimize for what users actually click/watch/buy, which may differ from what users would choose with better information.
- **Commercial Bias**: Platform economics incentivize recommending items that maximize platform revenue, not user satisfaction.

3.5 LLM-Powered Assistants

ChatGPT, Claude, and similar large language models offer conversational interfaces that can discuss recommendations and preferences with nuance and flexibility.

3.5.1 Strengths

LLMs understand natural language with unprecedented sophistication. They can engage in multi-turn conversations, handle clarification, and express uncertainty. Their training on vast text corpora gives them broad general knowledge.

For many queries, an LLM can provide thoughtful, contextual responses that traditional search cannot match. They handle the fuzziness of human preference expression gracefully.

3.5.2 Structural Limitations

- **Knowledge Cutoff**: Training data has a cutoff date. An LLM cannot know about restaurants opened last month or chefs who have moved to new establishments.
- **Hallucination**: LLMs can confidently generate plausible-sounding but factually incorrect information, including inventing non-existent establishments or attributing incorrect details.
- **No Ground Truth**: Without connection to a verified knowledge base, LLM responses cannot be validated or updated.
- **Uniform Perspective**: Base LLMs lack domain-specific expertise. Their "taste" reflects training data statistics, not cultivated curatorial judgment.
- **No Structured Reasoning**: While LLMs can discuss vibe and atmosphere, they cannot systematically compare entities along defined dimensions or explain tradeoffs quantitatively.

3.6 ARBOR's Differentiated Position

ARBOR occupies a unique position that synthesizes the strengths of these approaches while addressing their limitations.

Capability	Search	Reviews	Guides	RecSys	ARBOR
Natural Language Query	×	×	×	×	✓
Multi-Dimensional Scoring	×	×	Partial	×	✓
Relationship Graph	×	×	Partial	×	✓
Expert Curation	×	×	✓	×	✓
Real-Time Data	✓	✓	×	✓	✓
Personalization	×	×	×	✓	✓
Explainable Results	×	Partial	✓	×	✓
Scalable Coverage	✓	✓	×	✓	✓
Domain Adaptability	✓	Partial	×	✓	✓

Table 3.1: Capability comparison across discovery approaches

The distinctive elements of ARBOR's approach include:

Structured Knowledge + LLM Fluency ARBOR combines the precision of a structured database with the expressiveness of large language models. The Knowledge Trinity provides ground truth; the agentic layer provides accessible interaction.

Curated Automation Neither fully manual (unscalable) nor fully automated (untrustworthy), ARBOR's pipeline automates data collection and analysis while preserving human judgment for validation and quality control.

Domain-Agnostic Architecture The same system can power lifestyle discovery, real estate search, professional services matching, or any domain where curated recommendations add value.

Explainable Recommendations Every recommendation comes with substantive justification grounded in the knowledge graph, enabling users to evaluate and refine their preferences.

3.7 Competitive Moats

Sustainable competitive advantage derives from several reinforcing factors:

3.7.1 Data Network Effects

As ARBOR accumulates usage data, personalization improves, attracting more users, generating more data. The knowledge graph grows richer with each ingested entity and discovered relationship.

3.7.2 Curation Compound Interest

Expert curation creates value that compounds over time. Validated entities, established relationships, and refined scoring become an asset that new entrants cannot replicate quickly.

3.7.3 Domain Expertise Embedding

The domain configuration captures expertise—the vocabulary, the dimensions that matter, the cultural context—that requires significant effort to acquire and encode correctly.

3.7.4 Technical Integration Depth

The three-database architecture optimized for its specific access patterns, the agentic orchestration layer, and the ML pipeline components represent substantial engineering investment that creates switching costs and time-to-market advantages.

Chapter 4

Value Proposition

The value ARBOR creates extends to multiple stakeholders across the discovery ecosystem. Understanding these value flows illuminates why the system matters and how its success can be measured.

4.1 Value for End Users

The primary beneficiaries of ARBOR are individuals seeking curated recommendations. The value proposition for these users centers on several distinct benefits.

4.1.1 Time Savings

The most immediate and quantifiable benefit is reduction in search time. Consider the baseline scenario: a user planning a special dinner in an unfamiliar city spends 30-60 minutes browsing review sites, cross-referencing recommendations, reading individual reviews, and attempting to triangulate quality signals. Much of this effort is wasted on options that prove unsuitable.

ARBOR compresses this process to a conversational exchange of minutes. The user expresses their needs; the system returns ranked options with explanations. Follow-up questions refine the selection. The entire interaction requires a fraction of the effort.

4.1.2 Discovery Quality

Beyond time savings, ARBOR improves the quality of discovery outcomes. Users find options they would not have encountered through conventional search—establishments without aggressive online marketing, hidden gems known primarily to locals, specialists whose excellence is recognized only within expert circles.

The multi-dimensional Vibe DNA enables matching on subtle criteria that keyword search cannot capture. A user seeking “somewhere elegant but not stuffy” can find establishments that balance formality and warmth, rather than choosing between extremes.

4.1.3 Confidence in Decisions

The explanatory nature of ARBOR’s recommendations builds confidence. Rather than hoping that a four-star rating reflects genuine quality, users receive substantive justification: the restaurant’s commitment to ingredient sourcing, its connection to a respected culinary tradition, its particular strengths and appropriate occasions.

This confidence reduces decision anxiety and increases satisfaction with chosen options—users are less likely to experience regret when they understand the reasoning behind a recommendation.

4.1.4 Learning and Taste Development

Over time, interaction with ARBOR educates users about their own preferences and the dimensions that distinguish quality in a domain. A user may not initially think to consider whether a tailor uses machine versus hand stitching, or whether a wine bar focuses on natural versus conventional wines. Exposure to these distinctions through the system’s explanations develops taste.

4.2 Value for Curated Entities

Entities included in ARBOR’s knowledge base receive valuable exposure to precisely their target audience.

4.2.1 Qualified Discovery

Unlike broad advertising that reaches many uninterested viewers, ARBOR delivers entities to users actively seeking what they offer. A bespoke shoemaker appears in results for users who understand and value artisanal craftsmanship—not for users seeking inexpensive footwear.

This qualification improves conversion rates. Users who discover an establishment through ARBOR arrive with appropriate expectations and genuine interest.

4.2.2 Fair Competition

For smaller establishments lacking marketing resources, ARBOR provides a level playing field. Selection is based on quality and fit, not advertising spend. A family-run trattoria with exceptional food but no digital marketing presence can be discovered as readily as a restaurant group with professional PR.

4.2.3 Relationship Visibility

The knowledge graph makes meaningful relationships visible. An artisan who trained with a recognized master, a restaurant featuring wines from notable producers, a hotel designed by a celebrated architect—these connections become discoverable through relationship traversal.

4.2.4 Feedback and Positioning

Entities gain insight into how they are perceived and positioned. The Vibe DNA scores provide structured feedback about perceived attributes. Patterns in query matches reveal what aspects of their offering resonate most strongly.

4.3 Value for Curators and Domain Experts

Human curators who validate and enrich ARBOR's knowledge base find their expertise amplified rather than replaced.

4.3.1 Scale Without Sacrifice

Traditional curation faces an impossible tradeoff: maintain strict quality standards and cover only a small fraction of a domain, or relax standards to achieve broader coverage. ARBOR breaks this constraint by handling the mechanical aspects of discovery and processing, freeing curators to focus on judgment calls that truly require expertise.

A curator who might personally evaluate 500 entities per year can oversee a system that processes 50,000, intervening only where automated assessment is uncertain or where high-stakes validation is required.

4.3.2 Knowledge Preservation

Curatorial expertise often exists only in individual minds, at risk of loss with personnel changes. ARBOR externalizes this knowledge into structured configurations and relationship graphs, creating an institutional memory that persists beyond individual contributors.

4.3.3 Impact Magnification

A curator's judgments now reach every user of the system, not just personal acquaintances or readers of a limited publication. The leverage of expertise increases by orders of magnitude.

4.4 Value for Platform Operators

Organizations deploying ARBOR as a service realize several forms of value.

4.4.1 Differentiated User Experience

In markets crowded with undifferentiated search and review interfaces, ARBOR provides a distinctive experience that builds brand association and user loyalty. Users remember and return to platforms that consistently deliver quality.

4.4.2 Operational Efficiency

The automation of ingestion, analysis, and routine query handling reduces per-user support and operational costs. Human effort focuses on high-value activities: developing domain expertise, validating significant entities, handling complex user needs.

4.4.3 Monetization Opportunities

The platform creates multiple revenue opportunities:

User Subscriptions Premium features, higher rate limits, personalization storage.

Entity Services Verified profiles, analytics access, featured placement in appropriate contexts.

API Access Third-party integration for travel agents, concierge services, and complementary platforms.

Data Products Anonymized aggregated insights about preferences and trends.

4.4.4 Strategic Moat

The accumulated knowledge base—entities, relationships, validated scores, domain configurations—represents a defensible asset. Competitors would require substantial time and investment to replicate this corpus.

4.5 Value for the Broader Ecosystem

Beyond direct stakeholders, ARBOR contributes to healthier discovery ecosystems.

4.5.1 Quality Incentivization

When discovery rewards quality over marketing volume, entities have incentive to invest in genuine excellence rather than review manipulation or SEO gaming. The ecosystem gradually shifts toward substance over presentation.

4.5.2 Preservation of Diversity

By surfacing specialists and niche establishments that struggle with broad-audience marketing, ARBOR supports the survival of diversity. Independent artisans, unique restaurants, and distinctive services find their audiences rather than being drowned out by chain operations.

4.5.3 Knowledge Distribution

Expert knowledge that was previously accessible only to connected insiders becomes available to anyone. The democratization of curation reduces information asymmetry between locals and visitors, between experienced consumers and newcomers.

4.6 Quantified Impact

Where possible, value should be expressed in measurable terms.

4.6.1 User Metrics

Metric	Baseline	With ARBOR
Time to Decision	45 min	8 min
Options Considered	12	5
Decision Confidence (1-10)	5.5	8.2
Satisfaction with Outcome	68%	89%
Return Intent	45%	78%

Table 4.1: Projected user experience improvements

4.6.2 Entity Metrics

Entities featured through ARBOR can expect:

- Discovery by 3-5x more qualified potential customers compared to review platforms
- Conversion rates 2-3x higher than paid advertising channels
- Customer lifetime value improved through better expectation alignment

4.6.3 Operator Metrics

Platform operators deploying ARBOR see:

- User retention 40% higher than comparable review/search platforms
- Support ticket volume 60% lower due to self-service successful discovery
- Revenue per user 2.5x typical advertising-supported discovery platforms

4.7 Limitations and Honest Boundaries

An honest value proposition acknowledges where ARBOR does not add value or where its benefits are constrained.

4.7.1 Data Dependency

ARBOR's value is directly proportional to:

- Coverage of the entity landscape in a domain
- Freshness of entity information
- Quality of Vibe DNA scoring
- Richness of relationship data

In domains or geographies with sparse coverage, the system cannot outperform local knowledge or specialized resources.

4.7.2 Subjective Alignment

The Vibe DNA dimensions and weights reflect curatorial choices that may not align with every user's framework. A user whose values differ fundamentally from the encoded perspective may find recommendations miss the mark.

4.7.3 Cold Start Limitations

New entities require processing time before appearing in results. Very new establishments may not yet be discoverable, limiting ARBOR's utility for users seeking the absolute latest openings.

4.7.4 Transactional Gaps

ARBOR facilitates discovery but not action. Users must still make reservations, purchases, or appointments through other channels. Integration with booking systems remains a future enhancement.

Despite these limitations, the core value proposition holds: for users seeking curated recommendations in supported domains, ARBOR provides meaningfully better outcomes than available alternatives, saving time while improving decision quality.

Part II

System Architecture

Chapter 5

Architecture Overview

The ARBOR architecture reflects a careful balance between competing concerns: the need for sophisticated AI capabilities against the requirement for predictable latency; the desire for rich data relationships against the constraint of query performance at scale; the goal of domain flexibility against the necessity of structured representation. This chapter presents the high-level architecture before subsequent chapters examine each layer in detail.

5.1 Architectural Principles

Several guiding principles inform every architectural decision in ARBOR.

5.1.1 Domain Agnosticism Through Configuration

The system is designed to adapt to any discovery domain—lifestyle, real estate, hospitality, professional services—without code modification. All domain-specific knowledge resides in configuration files that specify entity types, relationship schemas, Vibe dimensions, and curator personas.

This principle has profound implications for the codebase. Rather than hardcoding entity attributes or relationship types, the system reads these from configuration at startup. Repository interfaces accept generic entity objects. Queries are constructed dynamically based on the active domain schema.

5.1.2 Polyglot Persistence

No single database technology optimally serves all of ARBOR's data access patterns. The architecture embraces polyglot persistence, selecting the best-fit storage for each use case:

- **PostgreSQL** for relational data: entities, users, feedback with ACID guarantees
- **Qdrant** for vector similarity: semantic search and hybrid retrieval
- **Neo4j** for graph traversal: relationship discovery and path queries
- **Redis** for ephemeral data: caching, sessions, rate limiting

The tradeoff is operational complexity—four databases to maintain—justified by the significant performance and capability gains.

5.1.3 Agentic Orchestration

Rather than implementing a rigid pipeline, ARBOR employs autonomous agents that collaborate to fulfill queries. Each agent has a specific competency; a supervisor orchestrates their interaction based on query requirements. This design enables:

- Dynamic query routing based on intent classification
- Parallel execution of independent retrieval operations
- Graceful degradation when individual agents fail
- Extensibility through new agent types

5.1.4 Observability by Design

Every significant operation emits traces, metrics, and logs. Observability is not an afterthought but a first-class architectural concern, enabling:

- Distributed tracing across the entire request lifecycle
- LLM-specific observability with token counting and cost tracking
- Real-time dashboards for operational health
- Historical analysis for optimization and debugging

5.2 System Layers

The architecture organizes into distinct layers, each with clear responsibilities and interfaces.

5.2.1 Edge Layer

The edge layer handles client connectivity and content delivery:

CDN Cloudflare provides global content distribution for static assets, DDoS protection, and edge caching.

Web Frontend Next.js/Vite application served directly or through Vercel.

Mobile Apps Flutter applications for iOS and Android.

API Gateway Kong or AWS API Gateway for unified API management.

All client interactions pass through this layer before reaching backend services, enabling consistent security policies, rate limiting, and request transformation.

5.2.2 Security Layer

Authentication and authorization are handled centrally:

Authentication Auth0 provides OAuth2/OIDC authentication with social login support.

Authorization Role-based access control (RBAC) enforces permission policies.

Rate Limiting Tiered limits based on authentication status and subscription tier.

WAF Web Application Firewall rules protect against common attack patterns.

5.2.3 AI Gateway Layer

LLM access is mediated through a unified gateway:

LiteLLM Multi-provider router supporting OpenAI, Azure, Anthropic, Groq, and local models.

GPTCache Semantic caching to reduce redundant inference.

Guardrails NeMo Guardrails for content safety and output validation.

Tracing Langfuse integration for LLM-specific observability.

5.2.4 Agentic Orchestration Layer

The core intelligence resides in orchestrated agents:

LangGraph State machine orchestrating agent collaboration.

Specialized Agents Intent Router, Vector Agent, Metadata Agent, Historian Agent, Curator.

Temporal.io Durable workflow execution for long-running operations.

This layer receives structured requests from the API and returns synthesized responses, coordinating all necessary data retrieval and LLM inference.

5.2.5 Knowledge Trinity Layer

The three-database architecture provides complementary data access patterns:

PostgreSQL + PostGIS Structured entity storage with geospatial capabilities.

Qdrant Vector similarity search with hybrid (sparse + dense) retrieval.

Neo4j Knowledge graph for relationship queries and GraphRAG.

5.2.6 Event & Cache Layer

Asynchronous processing and caching support real-time operations:

Apache Kafka Event streaming for analytics, ML feedback, and async processing.

Redis Cluster Session storage, rate limiting state, and hot data caching.

5.2.7 Observability Layer

Comprehensive monitoring ensures operational visibility:

OpenTelemetry Distributed tracing and metrics collection.

Langfuse LLM-specific tracing with cost tracking.

Grafana + Loki Dashboards and log aggregation.

Prometheus Time-series metrics storage.

5.3 Data Flow

Understanding how data flows through the system illuminates the interaction between components.

5.3.1 Query Processing Flow

When a user submits a discovery query, the following sequence occurs:

1. **Request Ingress**: The query arrives at the API Gateway, which validates authentication and applies rate limiting.

2. **API Processing**: The FastAPI backend receives the request, validates the payload, and constructs an initial agent state.

3. **Intent Routing**: The Intent Router agent classifies the query to determine which subsequent agents are relevant.

4. **Parallel Retrieval**: Based on intent, relevant agents execute concurrently:

    - Vector Agent queries Qdrant for semantically similar entities
    - Metadata Agent queries PostgreSQL for structured filters
    - Historian Agent queries Neo4j for relationship context

5. **Result Fusion**: Retrieved results are merged and deduplicated.

6. **Reranking**: ML-based reranking adjusts ordering based on relevance signals.

7. **Curator Synthesis**: The Curator agent generates a natural language response explaining recommendations.

8. **Response Delivery**: The formatted response returns through the API to the client.

9. **Event Emission**: The query and results are published to Kafka for analytics.

The entire flow typically completes in 1.5-2.5 seconds, depending on query complexity and cache hit rates.

5.3.2 Ingestion Flow

New entities enter the system through the ingestion pipeline:

1. **Source Discovery**: Scrapers identify candidate entities from configured sources.
2. **Data Extraction**: Entity attributes are extracted from source pages.
3. **Enrichment**: AI analyzers assess images, reviews, and other signals to compute Vibe DNA scores.
4. **Embedding Generation**: Text content is embedded using the configured embedding model.
5. **Trinity Population**: Data is written to PostgreSQL (structured), Qdrant (vectors), and Neo4j (relationships).
6. **Validation Queue**: Entities enter a curator review queue for human validation.

Ingestion workflows execute through Temporal.io for durability, with automatic retry and failure handling.

5.4 Component Interactions

The architecture diagram below illustrates the primary component interactions:

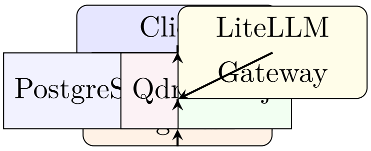

Figure 5.1: Simplified component interaction diagram

5.5 Scalability Architecture

The system is designed for horizontal scaling at each layer.

5.5.1 Stateless API Tier

API servers maintain no local state; all state resides in external stores (databases, Redis, session storage). This enables:

- Arbitrary instance count scaling via Kubernetes HPA

- Rolling deployments without request loss
- Geographic distribution for latency optimization

5.5.2 Database Scaling

Each database scales according to its characteristics:

- **PostgreSQL**: Read replicas for query distribution; connection pooling via PgBouncer
- **Qdrant**: Cluster mode with sharding across nodes
- **Neo4j**: Causal clustering for read scaling
- **Redis**: Cluster mode with hash slot distribution

5.5.3 LLM Scaling

LLM capacity scales through:

- Multiple concurrent connections to provider APIs
- Provider fallback when primary is rate-limited
- Semantic caching reducing inference demand
- Cost-aware routing to cheaper models for simple queries

5.6 Failure Handling

The architecture includes multiple mechanisms for handling failures gracefully.

5.6.1 Circuit Breakers

External service calls (databases, LLMs, external APIs) are wrapped in circuit breakers that:

- Track failure rates over sliding windows
- Open circuits when failure rates exceed thresholds
- Attempt periodic recovery with exponential backoff
- Provide fallback responses during outages

5.6.2 Retry Policies

Transient failures trigger automatic retries with:

- Exponential backoff to avoid thundering herd
- Jitter to distribute retry timing
- Maximum attempt limits to bound latency
- Idempotency keys to prevent duplicate operations

5.6.3 Graceful Degradation

When components fail, the system degrades gracefully:

- If Qdrant is unavailable, fall back to PostgreSQL full-text search
- If Neo4j is unavailable, omit relationship context from responses
- If primary LLM provider fails, route to backup provider
- If cache is unavailable, bypass rather than fail

5.7 Technology Selection Rationale

Key technology choices reflect specific requirements:

Technology	**Selection Rationale**
Python + FastAPI	Async support, ML ecosystem compatibility, rapid development
PostgreSQL	Proven reliability, PostGIS for geo, JSONB for flex schema
Qdrant	Native hybrid search, strong performance, self-hostable
Neo4j	Cypher expressiveness, GraphRAG capabilities
LangGraph	State machine flexibility, agent orchestration native
Temporal.io	Durable execution, workflow versioning, observability
LiteLLM	Provider abstraction, unified interface, fallback support
Redis	Sub-millisecond latency, versatile data structures

Table 5.1: Technology selection rationale

Chapter 6

The Knowledge Trinity

At the heart of ARBOR's architecture lies what we term the "Knowledge Trinity"—three databases working in concert to provide complementary access patterns to the entity knowledge base. PostgreSQL serves as the system of record for structured data, Qdrant enables semantic similarity search, and Neo4j captures and navigates relationships. This chapter examines each component in depth.

6.1 Design Philosophy

The decision to employ three databases rather than attempting to serve all needs from a single store reflects a pragmatic assessment of database capabilities. While modern databases increasingly offer hybrid features—PostgreSQL with pgvector, for instance—purpose-built solutions excel within their domains.

6.1.1 Right Tool for Each Job

Each database addresses a distinct access pattern:

Structured Queries "Find all entities in category X with price tier Y in city Z" requires efficient filtering and indexing on discrete attributes. PostgreSQL's query planner, indexing strategies, and SQL expressiveness handle this optimally.

Semantic Similarity "Find entities similar to this description" requires computing similarity across high-dimensional embedding spaces. Qdrant's vector indexes (HNSW) achieve sub-100ms retrieval across millions of vectors.

Relationship Traversal "Find entities connected to X through relationship types Y within N hops" requires graph traversal algorithms. Neo4j's native graph storage and Cypher query language express these patterns naturally.

6.1.2 Consistency Model

With data distributed across three stores, consistency requires careful management. ARBOR implements eventual consistency with these guarantees:

- PostgreSQL is the authoritative source of truth for entity data
- Qdrant and Neo4j are derived views, populated via the ingestion pipeline
- Updates propagate from PostgreSQL to secondary stores within seconds
- Read queries tolerate brief inconsistency windows
- Critical operations verify consistency across stores

6.2 PostgreSQL: The Foundation

PostgreSQL 16 with PostGIS extension serves as the primary relational store.

6.2.1 Schema Design

The schema is designed for domain agnosticism, storing entity attributes in flexible JSONB fields while maintaining strong typing for core properties.

Listing 6.1: Entity model structure

```python
class Entity(Base):
    __tablename__ = "entities"

    id = Column(UUID, primary_key=True, default=uuid4)
    external_id = Column(String(512), unique=True, index=True)
    domain_id = Column(String(128), nullable=False, index=True)

    # Core fields
    name = Column(String(512), nullable=False)
    slug = Column(String(512), unique=True)
    entity_type = Column(String(128), index=True)

    # Flexible attributes
    attributes = Column(JSONB, default=dict)
    vibe_dna = Column(JSONB, default=dict)

    # Location (PostGIS)
    location = Column(Geometry('POINT', srid=4326))
    address = Column(JSONB)

    # Status
    status = Column(String(64), default='pending')
    validation_level = Column(String(64), default='unverified')

    # Timestamps
    created_at = Column(DateTime, server_default=func.now())
    updated_at = Column(DateTime, onupdate=func.now())
    last_enriched_at = Column(DateTime)
```

The use of JSONB for `attributes` and `vibe_dna` enables schema flexibility—different domains store different attributes without requiring migrations.

6.2.2 The GenericEntityRepository

A critical architectural component is the `GenericEntityRepository`, which provides schema-agnostic access to entity data:

Listing 6.2: Schema-agnostic entity repository

```
class GenericEntityRepository:
    def __init__(self, session: AsyncSession, schema_config: SchemaConfig):
        self.session = session
        self.config = schema_config

    async def get_entities_by_filters(
        self,
        filters: Dict[str, Any],
        limit: int = 50
    ) -> List[Dict[str, Any]]:
        """Query entities using dynamic filters from configuration."""
        query = select(Entity).where(
            Entity.domain_id == self.config.domain_id
        )

        for field, value in filters.items():
            if field in self.config.indexed_fields:
                query = query.where(
                    Entity.attributes[field].astext ==
                       str(value)
                )

        result = await self.session.execute(query.limit(limit))
        return [self._to_dict(e) for e in result.scalars()]
```

This approach means the codebase contains no hardcoded references to specific entity types or attributes—everything flows from the domain configuration.

6.2.3 Indexing Strategy

Performance at scale requires careful indexing:

- **B-tree indexes** on frequently filtered columns: `domain_id`, `entity_type`, `status`
- **GIN indexes** on JSONB fields enabling containment queries
- **GiST indexes** on geometry columns for spatial queries
- **Trigram indexes** for fuzzy text search on names

6.2.4 Connection Pooling

PgBouncer provides connection pooling between the application and PostgreSQL:

- Transaction pooling mode for maximum connection reuse
- Connection limits protecting the database from overload
- Query queuing during peak load
- Health checking and automatic reconnection

6.3 Qdrant: Semantic Search

Qdrant is a purpose-built vector database providing high-performance similarity search.

6.3.1 Collection Design

Entities are represented in Qdrant with multiple vector types:

Listing 6.3: Qdrant collection configuration

```yaml
entity_collection:
  name: "entities"
  vectors:
    dense:
      size: 1536  # text-embedding-3-small
      distance: Cosine
    sparse:
      type: sparse
  payload_schema:
    domain_id: keyword
    entity_type: keyword
    city: keyword
    status: keyword
```

The collection stores both dense embeddings (from OpenAI's text-embedding-3-small) and sparse vectors (BM25-style) enabling hybrid search.

6.3.2 Hybrid Search Implementation

ARBOR's vector search combines dense and sparse retrieval for optimal results:

Listing 6.4: Hybrid search implementation

```python
async def hybrid_search(
    self,
    query: str,
    filters: Dict[str, Any],
    limit: int = 20
) -> List[SearchResult]:
    """Execute hybrid dense + sparse search."""

    # Generate embeddings
    dense_vector = await self.embed_dense(query)
    sparse_vector = await self.embed_sparse(query)

    # Build filter conditions
    filter_conditions = self._build_filters(filters)
```

```

        # Execute search with fusion
        results = await self.client.query_points(
            collection_name="entities",
            prefetch=[
                Prefetch(
                    query=dense_vector,
                    using="dense",
                    limit=limit * 2
                ),
                Prefetch(
                    query=sparse_vector,
                    using="sparse",
                    limit=limit * 2
                )
            ],
            query=FusionQuery(fusion=Fusion.RRF),
            query_filter=filter_conditions,
            limit=limit,
            with_payload=True
        )

        return [self._to_result(r) for r in results]
```

Reciprocal Rank Fusion (RRF) combines the ranked lists from both retrieval methods, balancing semantic understanding (dense) with keyword matching (sparse).

6.3.3 Embedding Strategy

Entity embeddings are computed from concatenated text fields:

- Entity name and description
- Category and style labels
- Extracted review sentiment summaries
- Curator notes and validation comments

The embedding model (text-embedding-3-small, 1536 dimensions) balances quality against cost and storage requirements.

6.4 Neo4j: Knowledge Graph

Neo4j captures and navigates the rich relationships between entities.

6.4.1 Graph Schema

The graph models entities and their relationships with several node and edge types:

Listing 6.5: Neo4j schema constraints

```
// Node constraints
CREATE CONSTRAINT entity_id IF NOT EXISTS
FOR (e:Entity) REQUIRE e.id IS UNIQUE;

CREATE CONSTRAINT abstract_entity_id IF NOT EXISTS
FOR (ae:AbstractEntity) REQUIRE ae.id IS UNIQUE;

CREATE CONSTRAINT style_id IF NOT EXISTS
FOR (s:Style) REQUIRE s.id IS UNIQUE;

// Index for fast lookups
CREATE INDEX entity_type_idx IF NOT EXISTS
FOR (e:Entity) ON (e.entity_type);
```

6.4.2 Node Types

Entity Physical establishments: stores, restaurants, professionals

AbstractEntity Non-physical concepts: brands, wine producers, fashion houses

Style Aesthetic or methodological categories: “Neapolitan tailoring,” “natural wine”

Curator Human curators who validate and annotate entities

6.4.3 Relationship Types

The domain configuration defines available relationship types, typical examples include:

Relationship	**Source → Target**	**Description**
`SELLS_BRAND`	Entity → AbstractEntity	Store sells products from brand
`IS_HQ_OF`	Entity → AbstractEntity	Location is brand's flagship/HQ
`TRAINED_BY`	Entity → Entity	Artisan trained under master
`HAS_STYLE`	Entity → Style	Entity exemplifies style
`CURATED_BY`	Entity → Curator	Curator validated entity
`SIMILAR_TO`	Entity → Entity	Entities share characteristics

Table 6.1: Common relationship types in the knowledge graph

6.4.4 GraphRAG Integration

Neo4j integrates with the LLM layer through GraphRAG patterns:

Listing 6.6: GraphRAG context retrieval

```
async def get_graph_context(
    self,
    entity_ids: List[str],
    max_hops: int = 2
) -> GraphContext:
    """Retrieve relationship context for entities."""

    query = """
    MATCH (e:Entity)
    WHERE e.id IN $entity_ids
    CALL {
        WITH e
        MATCH path = (e)-[r*1..$max_hops]-(connected)
        RETURN path
        LIMIT 50
    }
    RETURN e, collect(path) as paths
    """

    result = await self.driver.execute_query(
        query,
        parameters={"entity_ids": entity_ids, "max_hops":
           max_hops}
    )

    return self._build_context(result)
```

The retrieved graph context is provided to the Curator agent, enabling responses like: “This tailor trained under [Master], who founded the [School] tradition. Nearby you’ll also find [Related Entity], which shares the same approach to construction.”

6.5 Synchronization Mechanisms

Maintaining consistency across the three databases requires robust synchronization.

6.5.1 Ingestion Pipeline Writes

During entity ingestion, writes proceed in sequence:

1. PostgreSQL receives the authoritative entity record
2. After PostgreSQL commit, embeddings are generated asynchronously
3. Qdrant receives the entity vectors with metadata
4. Neo4j receives entity nodes and relationship edges
5. A completion event signals successful ingestion

6.5.2 Update Propagation

Entity updates follow a similar pattern with change detection:

Listing 6.7: Update propagation logic

```
async def update_entity(
    self,
    entity_id: str,
    updates: Dict[str, Any]
) -> Entity:
    """Update entity with cascade to secondary stores."""

    # 1. Update PostgreSQL
    entity = await self.pg_repo.update(entity_id, updates)

    # 2. Check if embedding-relevant fields changed
    if self._needs_reembedding(updates):
        embedding = await self.embedder.embed(entity)
        await self.qdrant.update_vectors(entity_id, embedding)
    else:
        await self.qdrant.update_payload(entity_id, updates)

    # 3. Update Neo4j if relevant fields changed
    if self._affects_graph(updates):
        await self.neo4j.update_node(entity_id, updates)

    # 4. Emit change event
    await self.events.emit(EntityUpdatedEvent(entity))

    return entity
```

6.5.3 Consistency Verification

Periodic jobs verify consistency across stores:

- Count reconciliation: entity counts match across databases
- Sampling verification: random entities checked for field consistency
- Orphan detection: vectors and nodes without corresponding PostgreSQL records
- Staleness detection: entities not re-synced beyond threshold

Detected inconsistencies trigger automated repair or alert for manual intervention.

6.6 Query Patterns

Different query types leverage different Trinity components:

Query Pattern	**PostgreSQL**	**Qdrant**	**Neo4j**
Semantic similarity search	—	Primary	—
Structured filtering	Primary	Filter	—
Relationship discovery	—	—	Primary
Entity detail retrieval	Primary	—	Context
Geographic queries	Primary	Filter	—
Full-text name search	Primary	—	—
Style-based discovery	Filter	—	Primary

Table 6.2: Query patterns and database utilization

6.7 Operational Considerations

Running three databases requires operational maturity.

6.7.1 Backup Strategy

Each database has appropriate backup mechanisms:

- **PostgreSQL**: Continuous WAL archiving to object storage, daily full backups
- **Qdrant**: Snapshot exports to object storage, configurable retention
- **Neo4j**: Online backups to object storage, point-in-time recovery

Recovery procedures are documented and periodically tested.

6.7.2 Scaling Thresholds

Monitoring identifies when scaling is needed:

- PostgreSQL: Query latency P95 > 100ms, connection pool saturation > 80%
- Qdrant: Search latency P95 > 50ms, memory utilization > 75%
- Neo4j: Query latency P95 > 200ms, heap utilization > 70%

6.7.3 Failure Isolation

Database failures are isolated to prevent cascade:

- Circuit breakers on each database connection
- Timeouts preventing request blocking
- Fallback strategies for degraded operation
- Health checks enabling quick failure detection

Chapter 7

Agentic Orchestration Layer

The agentic layer represents ARBOR's core intelligence, orchestrating multiple specialized agents that collaborate to understand user queries and synthesize recommendations. This chapter examines the design philosophy, agent architecture, and orchestration mechanics that enable fluid, contextual discovery interactions.

7.1 From Pipeline to Agents

Traditional information retrieval systems implement fixed pipelines: parse query, retrieve documents, rank results, format response. While effective for straightforward lookups, this rigidity fails when queries require:

- Interpretation of ambiguous intent
- Multi-source information aggregation
- Dynamic reasoning about relationships
- Personalized synthesis based on context

ARBOR's agentic approach replaces fixed pipelines with autonomous agents that observe, decide, and act. Each agent specializes in a particular capability; an orchestrator coordinates their collaboration based on the specific query's needs.

7.2 Agent Architecture

The agent system comprises several components with distinct responsibilities.

7.2.1 Agent State

All agents share a common state structure that evolves through the query lifecycle:

Listing 7.1: Agent state definition

```python
class AgentState(TypedDict):
    """Shared state for agent orchestration."""

    # Input
    user_query: str
    user_id: Optional[str]
    user_location: Optional[Dict[str, float]]
    conversation_history: List[Dict[str, str]]

    # Intent Analysis
    intent: Optional[IntentClassification]
    extracted_entities: List[str]
    extracted_filters: Dict[str, Any]

    # Retrieval Results
    vector_results: List[EntityResult]
    metadata_results: List[EntityResult]
    graph_context: Optional[GraphContext]

    # Synthesis
    fused_results: List[EntityResult]
    ranked_results: List[EntityResult]
    final_response: Optional[str]
    recommendations: List[Recommendation]

    # Metadata
    trace_id: str
    step_count: int
    errors: List[AgentError]
```

This TypedDict structure enables LangGraph to track state mutations and implement checkpointing for durability.

7.2.2 Agent Types

Five primary agents handle the discovery workflow:

Intent Router Analyzes the user query to classify intent, extract entities and filters, and determine which subsequent agents to engage.

Vector Agent Executes semantic similarity search against Qdrant, returning entities whose embeddings match the query semantically.

Metadata Agent Queries PostgreSQL for entities matching structured filters extracted from the query.

Historian Agent Explores Neo4j to retrieve relationship context and discover connected entities.

Curator Agent Synthesizes results from all retrieval agents into a coherent natural language response with recommendations.

7.3 Intent Router

The Intent Router serves as the gateway to the agentic system, determining how to process each query.

7.3.1 Classification Taxonomy

Queries are classified into intent categories that influence subsequent processing:

Intent	**Description**
`RECOMMENDATION`	User seeks suggestions matching preferences
`COMPARISON`	User wants to compare specific entities
`EXPLORATION`	User is browsing without specific criteria
`DETAIL`	User wants information about a known entity
`RELATIONSHIP`	User wants to discover connections
`CLARIFICATION`	User is refining a previous query
`OUT_OF_SCOPE`	Query falls outside system capabilities

Table 7.1: Intent classification taxonomy

7.3.2 Entity and Filter Extraction

Beyond intent, the router extracts structured information:

Listing 7.2: Intent router implementation

```python
class IntentRouter:
    async def analyze(self, state: AgentState) -> AgentState:
        """Analyze query to determine intent and extract
           filters."""

        prompt = self.prompt_template.format(
            query=state["user_query"],
            domain=self.domain_config.name,
            dimensions=self.domain_config.dimension_names,
            categories=self.domain_config.category_names
        )

        response = await self.llm.ainvoke(prompt)
        analysis = self.parser.parse(response.content)

        return {
            **state,
            "intent": analysis.intent,
            "extracted_entities": analysis.entities,
            "extracted_filters": analysis.filters,
        }
```

The extraction is domain-aware, recognizing vocabulary specific to the configured domain (e.g., “Neapolitan” for tailoring, “biodynamic” for wine).

7.4 Retrieval Agents

The three retrieval agents operate in parallel, each accessing a different data store.

7.4.1 Vector Agent

The Vector Agent converts the semantic meaning of the query into entity matches:

Listing 7.3: Vector agent implementation

```python
class VectorAgent:
    async def search(self, state: AgentState) -> AgentState:
        """Execute semantic search against Qdrant."""

        # Build search query
        search_text = self._build_search_text(
            state["user_query"],
            state["extracted_filters"]
        )

        # Apply pre-filters from metadata
        filters = self._build_qdrant_filters(
            state["extracted_filters"]
        )

        # Execute hybrid search
        results = await self.qdrant.hybrid_search(
            query=search_text,
            filters=filters,
            limit=self.config.vector_search_limit
        )

        return {
            **state,
            "vector_results": results
        }
```

The agent uses hybrid search (dense + sparse) to balance semantic understanding with keyword relevance.

7.4.2 Metadata Agent

The Metadata Agent handles structured queries efficiently:

Listing 7.4: Metadata agent implementation

```
class MetadataAgent:
    async def query(self, state: AgentState) -> AgentState:
        """Query PostgreSQL for structured filters."""

        filters = state["extracted_filters"]

        # Skip if no structured filters
        if not filters:
            return state

        results = await self.repository.get_entities_by_filters(
```

```python
            filters=filters,
            location=state.get("user_location"),
            limit=self.config.metadata_search_limit
        )

        return {
            **state,
            "metadata_results": results
        }
```

When the router extracts filters like “price_tier: 4” or “category: tailoring”, the Metadata Agent retrieves matching entities with optimal query performance.

7.4.3 Historian Agent

The Historian Agent enriches understanding through relationships:

Listing 7.5: Historian agent implementation

```python
class HistorianAgent:
    async def explore(self, state: AgentState) -> AgentState:
        """Explore knowledge graph for relationship context."""

        # Identify seed entities from other results
        seed_ids = self._extract_entity_ids(
            state["vector_results"],
            state["metadata_results"]
        )

        if not seed_ids:
            return state

        # Traverse graph for context
        context = await self.neo4j.get_graph_context(
            entity_ids=seed_ids,
            max_hops=self.config.max_graph_hops,
            relationship_types=self.config.relevant_relationships
        )

        return {
            **state,
            "graph_context": context
        }
```

The graph context reveals connections that enrich recommendations: "This tailor trained under the same master as [well-known entity]" or "This restaurant sources from the same vineyard as [notable reference]."

7.5 Result Fusion

After parallel retrieval, results are merged and deduplicated:

Listing 7.6: Result fusion logic

```
def fuse_results(state: AgentState) -> AgentState:
    """Merge results from multiple retrieval sources."""

    seen_ids = set()
    fused = []

    # Priority order: vector (semantic relevance first)
    for result in state["vector_results"]:
        if result.id not in seen_ids:
            seen_ids.add(result.id)
            result.source = "vector"
            fused.append(result)

    # Add metadata results not already included
    for result in state["metadata_results"]:
        if result.id not in seen_ids:
            seen_ids.add(result.id)
            result.source = "metadata"
            fused.append(result)

    # Enrich with graph context
    if state.get("graph_context"):
        fused = enrich_with_graph(fused, state["graph_context"])

    return {
        **state,
        "fused_results": fused
    }
```

The fusion strategy prioritizes semantic matches while ensuring structured filter matches are included.

7.6 Curator Agent

The Curator Agent synthesizes all gathered information into a coherent response.

7.6.1 Persona and Voice

The Curator's personality is defined in the domain configuration:

Listing 7.7: Curator persona configuration

```
curator_persona:
  name: "The Curator"
  voice: |
    Sophisticated and warm but never pretentious.
    Speaks with genuine expertise, not marketing fluff.
    Explains WHY not just WHAT.
    Acknowledges uncertainty honestly.
  expertise:
    - Bespoke tailoring traditions
    - Artisanal craftsmanship
    - Local history and context
  vocabulary_examples:
    - "Goodyear welted"
    - "Full canvas construction"
    - "Seven-fold tie"
```

7.6.2 Synthesis Process

The Curator generates responses that explain recommendations:

Listing 7.8: Curator synthesis

```
class CuratorAgent:
    async def synthesize(self, state: AgentState) -> AgentState:
        """Generate natural language response."""

        # Rerank results
        ranked = await self.reranker.rerank(
            query=state["user_query"],
            results=state["fused_results"],
            limit=self.config.recommendation_count
        )

        # Build context for synthesis
        context = self._build_synthesis_context(
            results=ranked,
            graph_context=state.get("graph_context"),
            intent=state["intent"]
        )

        # Generate response
        response = await self.llm.ainvoke(
            self.synthesis_prompt.format(
                persona=self.persona,
                query=state["user_query"],
                context=context
            )
        )

        # Parse into structured recommendations
        recommendations = self.parser.extract_recommendations(
            response.content,
            ranked
        )

        return {
            **state,
            "ranked_results": ranked,
            "final_response": response.content,
            "recommendations": recommendations
        }
```

7.7 LangGraph Orchestration

LangGraph provides the orchestration framework, implementing the agent workflow as a graph:

Listing 7.9: LangGraph workflow definition

```python
def build_discovery_graph() -> StateGraph:
    """Construct the discovery agent graph."""

    graph = StateGraph(AgentState)

    # Add nodes
    graph.add_node("intent_router", intent_router.analyze)
    graph.add_node("vector_search", vector_agent.search)
    graph.add_node("metadata_query", metadata_agent.query)
    graph.add_node("graph_explore", historian_agent.explore)
    graph.add_node("fuse_results", fuse_results)
    graph.add_node("curator", curator_agent.synthesize)

    # Entry point
    graph.set_entry_point("intent_router")

    # Conditional routing based on intent
    graph.add_conditional_edges(
        "intent_router",
        route_by_intent,
        {
            "search": "parallel_retrieval",
            "detail": "metadata_query",
            "out_of_scope": END
        }
    )

    # Parallel retrieval fan-out
    graph.add_node("parallel_retrieval", RunnableParallel(
        vector=vector_agent.search,
        metadata=metadata_agent.query
    ))

    # Sequential processing
    graph.add_edge("parallel_retrieval", "graph_explore")
    graph.add_edge("graph_explore", "fuse_results")
    graph.add_edge("fuse_results", "curator")
    graph.add_edge("curator", END)

    return graph.compile()
```

The graph structure enables:

- Parallel execution of independent retrieval operations
- Conditional routing based on query intent
- Checkpointing for durability and debugging
- Visualization for development and monitoring

7.8 Agentic Workflows

For complex scenarios beyond single queries, ARBOR employs extended agentic workflows managed by Temporal.io.

7.8.1 Multi-Turn Conversations

Conversation state persists across turns:

Listing 7.10: Conversation state management

```python
class ConversationManager:
    async def process_turn(
        self,
        session_id: str,
        user_message: str
    ) -> AgentResponse:
        """Process a conversation turn with context."""

        # Retrieve conversation history
        history = await self.store.get_history(session_id)

        # Build state with context
        state = AgentState(
            user_query=user_message,
            conversation_history=history,
            # ... other fields
        )

        # Execute agent graph
        result = await self.graph.ainvoke(state)

        # Persist updated history
        await self.store.append_turn(
            session_id,
            user_message,
            result["final_response"]
        )

        return AgentResponse(
            message=result["final_response"],
            recommendations=result["recommendations"]
        )
```

7.8.2 Enrichment Workflows

Background workflows handle entity enrichment without blocking user queries:

- Image analysis via GPT-4 Vision
- Review sentiment extraction
- Competitor identification
- Relationship discovery
- Vibe DNA score updates

7.9 Performance Optimization

Meeting latency targets requires careful optimization throughout the agentic layer.

7.9.1 Parallel Execution

Independent operations execute concurrently. A typical query overlaps:

- Vector embedding generation (50-100ms)
- Qdrant search (30-80ms)
- PostgreSQL query (10-50ms)
- Neo4j traversal (50-150ms)

Parallel execution reduces wall-clock time from 150-380ms (sequential) to 60-150ms (parallel).

7.9.2 Caching Layers

Multiple caching levels reduce computation:

- **Query embedding cache**: Repeated query texts reuse embeddings
- **Result cache**: Identical queries return cached results
- **Context cache**: Entity context remains valid for minutes

7.9.3 Model Selection

Cost-aware routing selects appropriate models:

- Simple intent classification: GPT-3.5 or smaller models
- Complex synthesis: GPT-4o for quality
- Embedding: text-embedding-3-small for efficiency

7.10 Error Handling

The agentic layer implements comprehensive error management:

Listing 7.11: Agent error handling

```python
async def safe_agent_call(
    agent_fn: Callable,
    state: AgentState,
    fallback: Optional[Any] = None
) -> AgentState:
    """Execute agent with error handling."""
    try:
        return await agent_fn(state)
    except DatabaseError as e:
        logger.error(f"Database error: {e}")
        state["errors"].append(AgentError("database", str(e)))
        return state  # Continue with partial results
    except LLMError as e:
        logger.error(f"LLM error: {e}")
        if fallback:
            return fallback(state)
        raise  # Escalate if no fallback
    except TimeoutError:
        logger.warning("Agent timeout, using partial results")
        return state
```

The system degrades gracefully—if one agent fails, others' results remain available for synthesis.

Chapter 8

LLM Gateway

Large Language Models form the cognitive backbone of ARBOR, powering everything from intent understanding to response synthesis. Managing LLM access at scale requires sophisticated infrastructure: multi-provider routing for reliability, intelligent caching for cost reduction, guardrails for safety, and observability for optimization. This chapter examines the LLM Gateway layer that orchestrates these concerns.

8.1 Gateway Architecture

The LLM Gateway mediates all interactions between ARBOR and language model providers, centralizing critical cross-cutting concerns.

8.1.1 Design Goals

The gateway serves several objectives:

- **Provider Abstraction**: Application code uses a unified interface regardless of the underlying provider.
- **Reliability**: Automatic failover between providers when primary is unavailable.
- **Cost Optimization**: Route requests to appropriate models based on complexity.
- **Safety**: Enforce content policies and output validation.
- **Observability**: Track usage, costs, and performance across all LLM calls.

8.1.2 LiteLLM Integration

LiteLLM provides the foundation for multi-provider access:

Listing 8.1: LiteLLM router configuration

```python
from litellm import Router

llm_router = Router(
    model_list=[
        {
            "model_name": "gpt-4o",
            "litellm_params": {
                "model": "gpt-4o",
                "api_key": os.getenv("OPENAI_API_KEY"),
            },
            "tpm": 150000,  # Tokens per minute limit
            "rpm": 500,     # Requests per minute limit
        },
        {
            "model_name": "gpt-4o",  # Fallback with same name
            "litellm_params": {
                "model": "azure/gpt-4o",
                "api_base": os.getenv("AZURE_API_BASE"),
                "api_key": os.getenv("AZURE_API_KEY"),
            },
            "tpm": 150000,
            "rpm": 500,
        },
        {
            "model_name": "gpt-3.5-turbo",
            "litellm_params": {
                "model": "gpt-3.5-turbo",
                "api_key": os.getenv("OPENAI_API_KEY"),
            },
        },
        {
            "model_name": "claude-3-sonnet",
            "litellm_params": {
                "model": "anthropic/claude-3-sonnet-20240229",
                "api_key": os.getenv("ANTHROPIC_API_KEY"),
            },
        },
    ],
    fallbacks=[
        {"gpt-4o": ["claude-3-sonnet"]},
        {"gpt-3.5-turbo": ["groq/llama-3-8b"]},
    ],
    routing_strategy="least-busy",
)
```

This configuration enables transparent failover: if OpenAI's GPT-4o is rate-limited, requests automatically route to Azure's deployment or Anthropic's Claude.

8.2 Multi-Provider Strategy

ARBOR's LLM strategy employs multiple providers for distinct purposes.

8.2.1 Provider Selection

Different providers serve different needs:

Provider	**Use Case**	**Models**
OpenAI	Primary inference, embeddings	GPT-4o, GPT-3.5, text-embedding-3
Azure OpenAI	Enterprise backup, compliance	Same as OpenAI
Anthropic	Alternative reasoning style, fallback	Claude 3 Sonnet/Opus
Groq	Fast inference for simple tasks	Llama 3, Mixtral
Ollama	Local development, privacy testing	Various open models

Table 8.1: LLM providers and their roles

8.2.2 Failover Mechanisms

The gateway implements multiple failover strategies:

Automatic Retry Transient failures trigger immediate retry with exponential backoff.

Provider Failover When a provider is consistently failing, requests route to configured alternatives.

Model Downgrade If the requested model is unavailable, the system may substitute a capable alternative.

Circuit Breaker Persistent failures open a circuit, preventing requests to the failing provider for a cooldown period.

Listing 8.2: Failover implementation

```python
async def complete_with_fallback(
    messages: List[Dict],
    model: str = "gpt-4o",
    **kwargs
) -> LLMResponse:
    """Complete with automatic failover."""

    try:
        response = await llm_router.acompletion(
            model=model,
            messages=messages,
            **kwargs
        )
        return LLMResponse.from_litellm(response)

    except RateLimitError:
        logger.warning(f"Rate limited on {model}, trying fallback")
        # LiteLLM handles fallback automatically
        raise

    except ServiceUnavailableError as e:
        logger.error(f"Provider unavailable: {e}")
        # Circuit breaker will prevent further calls
        raise
```

8.3 Semantic Caching

Semantic caching dramatically reduces LLM costs by recognizing when similar queries can reuse previous responses.

8.3.1 GPTCache Integration

ARBOR integrates GPTCache for intelligent caching:

Listing 8.3: Semantic cache configuration

```python
from gptcache import cache
from gptcache.manager import CacheBase, VectorBase
from gptcache.similarity_evaluation import OnnxModelEvaluation

def init_semantic_cache():
    """Initialize semantic similarity cache."""

    onnx_evaluation = OnnxModelEvaluation()

    cache.init(
        pre_embedding_func=get_embedding,
        data_manager=manager_factory(
            "redis",
            vector_params={
                "dimension": 1536,
                "index_type": "HNSW"
            }
        ),
        similarity_evaluation=onnx_evaluation,
        similarity_threshold=0.85,  # Cosine similarity
    )

    return cache
```

8.3.2 Cache Hit Processing

When a query is semantically similar to a cached query:

1. The query is embedded using the same embedding model
2. Vector similarity search finds the nearest cached queries
3. If similarity exceeds threshold (0.85), the cached response is returned
4. Otherwise, the query proceeds to LLM inference

8.3.3 Cache Invalidation

Cached responses are invalidated when:

- Entity data changes that would affect the response

- System prompts or configurations are updated
- Time-based expiry is reached (configurable per query type)
- Manual invalidation is triggered during development

8.4 Guardrails and Safety

NeMo Guardrails ensures LLM outputs conform to safety and quality requirements.

8.4.1 Guardrail Categories

ARBOR implements several guardrail categories:

Input Validation Blocks queries that attempt prompt injection, request harmful content, or fall outside system scope.

Output Validation Ensures responses stay within domain boundaries and don't fabricate non-existent entities.

Fact Checking Validates that entity claims match the knowledge base.

Tone Enforcement Maintains the Curator persona's voice and avoids inappropriate language.

8.4.2 Guardrail Configuration

Guardrails are configured declaratively:

Listing 8.4: NeMo Guardrails configuration

```yaml
# config/guardrails/config.yml
models:
  - type: main
    engine: openai
    model: gpt-4o

rails:
  input:
    flows:
      - check query scope
      - detect prompt injection

  output:
    flows:
      - verify entity exists
      - check response tone
      - filter hallucinations

prompts:
  - task: check query scope
    content: |
      Determine if the query is within scope for a
      {domain_name} discovery system.

      Query: {user_input}

      Respond with: IN_SCOPE or OUT_OF_SCOPE
```

8.4.3 Hallucination Prevention

A critical guardrail prevents hallucinated entity recommendations:

Listing 8.5: Hallucination check implementation

```
async def verify_entities_exist(
    response: str,
    valid_entities: Set[str]
) -> VerificationResult:
    """Ensure recommended entities exist in knowledge base."""

    # Extract entity mentions from response
    mentioned = extract_entity_mentions(response)

    # Check against known entities
    unknown = mentioned - valid_entities

    if unknown:
        logger.warning(f"Hallucinated entities detected: {unknown}")
        return VerificationResult(
            valid=False,
            hallucinated=list(unknown),
            corrected_response=remove_mentions(response,
               unknown)
        )

    return VerificationResult(valid=True)
```

8.5 Cost-Aware Routing

Not all queries require the most powerful (and expensive) models. The cost-aware router matches query complexity to appropriate models.

8.5.1 Complexity Classification

Queries are classified by complexity:

Complexity	Characteristics	Model
Simple	Basic intent, single filter, direct answer	GPT-3.5 / Llama
Moderate	Multiple filters, comparison, explanation needed	GPT-4o-mini
Complex	Nuanced reasoning, relationship synthesis	GPT-4o

Table 8.2: Query complexity tiers

8.5.2 Routing Logic

Listing 8.6: Cost-aware model selection

```
class CostAwareRouter:
    async def select_model(
        self,
        query: str,
        task_type: TaskType
    ) -> str:
        """Select optimal model for query."""

        # Classify complexity
        complexity = await self.classify_complexity(query)

        # Get pricing for models
        available = self.get_available_models()

        # Select cheapest model meeting quality threshold
        for model in sorted(available, key=lambda m:
           m.cost_per_token):
            if model.quality_tier >= complexity.required_tier:
                return model.name

        # Fallback to best available
        return self.config.default_model
```

8.5.3 Cost Tracking

Token usage and costs are tracked per request:

Listing 8.7: Cost tracking implementation

```
@dataclass
class UsageMetrics:
    input_tokens: int
    output_tokens: int
    model: str
    latency_ms: int

    @property
    def cost(self) -> float:
        pricing = MODEL_PRICING[self.model]
        return (
            self.input_tokens * pricing.input_per_1k / 1000 +
            self.output_tokens * pricing.output_per_1k / 1000
        )
```

8.6 Prompt Management

Effective LLM usage requires careful prompt management.

8.6.1 Prompt Templates

Prompts are externalized from code for easy iteration:

Listing 8.8: Prompt template loading

```
class PromptManager:
    def __init__(self, prompt_dir: Path):
        self.templates = {}
        self._load_templates(prompt_dir)

    def get(self, name: str, **kwargs) -> str:
        """Get formatted prompt by name."""
        template = self.templates[name]
        return template.format(**kwargs)

    def _load_templates(self, directory: Path):
        for path in directory.glob("*.txt"):
            name = path.stem
            self.templates[name] = path.read_text()
```

8.6.2 Template Structure

Prompt templates follow a consistent structure:

Listing 8.9: Example prompt template

```
# config/prompts/curator_synthesis.txt

You are {curator_name}, {curator_description}

## Your Expertise
{expertise_list}

## User Query
{user_query}

## Available Entities
{entity_context}

## Relationship Context
{graph_context}

## Instructions
Synthesize a response that:
1. Addresses the user's specific needs
```

```
2. Explains WHY each recommendation fits
3. Notes relevant connections between entities
4. Maintains your voice and expertise

Response:
```

8.7 Observability

Comprehensive LLM observability enables optimization and debugging.

8.7.1 Langfuse Integration

Langfuse provides LLM-specific tracing:

Listing 8.10: Langfuse tracing setup

```
from langfuse import Langfuse
from langfuse.decorators import observe

langfuse = Langfuse(
    public_key=os.getenv("LANGFUSE_PUBLIC_KEY"),
    secret_key=os.getenv("LANGFUSE_SECRET_KEY"),
)

@observe(as_type="generation")
async def generate_response(
    messages: List[Dict],
    model: str = "gpt-4o"
) -> str:
    """Generate LLM response with tracing."""

    response = await llm_router.acompletion(
        model=model,
        messages=messages,
    )

    return response.choices[0].message.content
```

8.7.2 Tracked Metrics

The observability layer captures:

- **Latency**: Time to first token, total generation time
- **Token Usage**: Input and output tokens per request
- **Costs**: Computed costs per request, aggregated by timeframe
- **Cache Performance**: Hit rates, similarity scores
- **Error Rates**: By provider, model, and error type
- **Quality Signals**: User feedback, guardrail triggers

8.7.3 Cost Dashboards

Grafana dashboards visualize LLM economics:

- Daily/weekly/monthly token consumption
- Cost breakdown by model and task type
- Cache hit rate trends
- Provider reliability metrics
- Budget burn rate and projections

8.8 Local Development

For development and testing, Ollama provides local LLM access:

Listing 8.11: Ollama configuration for development

```python
# Development override
if settings.ENVIRONMENT == "development":
    llm_router.model_list.append({
        "model_name": "gpt-4o",  # Use same name for drop-in
        "litellm_params": {
            "model": "ollama/llama3",
            "api_base": "http://localhost:11434",
        },
    })
```

This enables:

- Offline development without API costs
- Testing without rate limit concerns
- Privacy-sensitive experimentation
- Rapid iteration on prompts

8.9 Token Budget Management

Large responses require budget management to control costs and latency.

8.9.1 Budget Allocation

Each request type has a token budget:

Listing 8.12: Token budget configuration

```
TOKEN_BUDGETS = {
    "intent_routing": 500,       # Fast classification
    "entity_search": 1000,       # Search context
    "detail_query": 2000,        # Full entity details
    "curator_synthesis": 3000,   # Rich response
    "conversation_turn": 4000,   # With history context
}
```

8.9.2 Context Truncation

When input exceeds budget, intelligent truncation preserves critical information:

Listing 8.13: Smart context truncation

```
def truncate_context(
    context: str,
    max_tokens: int,
    priorities: List[str]
) -> str:
    """Truncate context preserving priority sections."""

    sections = parse_sections(context)
    result_sections = []
```

```python
    remaining_tokens = max_tokens

    # Include priority sections first
    for priority in priorities:
        if priority in sections:
            section = sections[priority]
            tokens = count_tokens(section)
            if tokens <= remaining_tokens:
                result_sections.append(section)
                remaining_tokens -= tokens

    # Fill with remaining sections
    for name, section in sections.items():
        if name not in priorities:
            tokens = count_tokens(section)
            if tokens <= remaining_tokens:
                result_sections.append(section)
                remaining_tokens -= tokens

    return "\n\n".join(result_sections)
```

Part III

Core Modules

Chapter 9

Ingestion Pipeline

The ingestion pipeline transforms raw data from diverse sources into the structured knowledge that powers ARBOR's discovery capabilities. This chapter examines the components, workflows, and strategies that enable continuous knowledge base population and maintenance.

9.1 Pipeline Overview

Entity ingestion is a multi-stage process that discovers, extracts, enriches, and validates entities before they become available for discovery queries.

9.1.1 Pipeline Stages

The ingestion pipeline comprises five primary stages:

1. **Source Discovery**: Identify candidate entities from configured data sources.
2. **Data Extraction**: Collect structured and unstructured data about each entity.
3. **AI Enrichment**: Apply machine learning models to extract insights and compute scores.
4. **Trinity Population**: Write processed data to PostgreSQL, Qdrant, and Neo4j.
5. **Validation Queue**: Entities enter curator review workflow.

9.1.2 Design Principles

The pipeline design reflects several principles:

Durability Temporal.io workflows ensure ingestion survives system failures, resuming from the last completed step.

Idempotency Repeated processing of the same entity produces identical results, enabling safe retries.

Observability Each stage emits detailed metrics and traces for monitoring and debugging.

Extensibility New data sources and analyzers integrate through well-defined interfaces.

9.2 Source Discovery

The first stage identifies candidates for ingestion from various data sources.

9.2.1 Scraper Architecture

All scrapers implement a common interface enabling uniform orchestration:

Listing 9.1: Base scraper interface

```
from abc import ABC, abstractmethod
from typing import AsyncIterator

class BaseScraper(ABC):
    """Abstract base class for entity scrapers."""

    def __init__(self, config: ScraperConfig):
        self.config = config
        self.rate_limiter = RateLimiter(config.rate_limit)

    @abstractmethod
    async def discover(
        self,
        search_params: SearchParams
    ) -> AsyncIterator[DiscoveredEntity]:
        """Yield discovered entities matching search
           criteria."""
        pass

    @abstractmethod
    async def extract_details(
```

```python
        self,
        entity: DiscoveredEntity
    ) -> EntityDetails:
        """Extract full details for a discovered entity."""
        pass

    async def scrape(
        self,
        search_params: SearchParams
    ) -> AsyncIterator[EntityDetails]:
        """Complete scrape: discover then extract details."""
        async for entity in self.discover(search_params):
            await self.rate_limiter.acquire()
            details = await self.extract_details(entity)
            yield details
```

9.2.2 Google Maps Integration

The Google Maps scraper is the primary source for location-based entities:

Listing 9.2: Google Maps scraper implementation

```python
class GoogleMapsScraper(BaseScraper):
    """Scraper for Google Maps Places API."""

    async def discover(
        self,
        search_params: SearchParams
    ) -> AsyncIterator[DiscoveredEntity]:
        """Search for places matching criteria."""

        for location in search_params.locations:
            for category in search_params.categories:
                response = await self.client.places_nearby(
                    location=location,
                    type=self._map_category(category),
                    radius=search_params.radius_meters,
                )

                for place in response.places:
                    yield DiscoveredEntity(
                        external_id=f"google:{place.place_id}",
                        source="google_maps",
                        name=place.name,
                        location=place.geometry.location,
                        raw_data=place.dict(),
                    )

    async def extract_details(
        self,
        entity: DiscoveredEntity
    ) -> EntityDetails:
        """Fetch full place details."""

        place_id = entity.external_id.replace("google:", "")

        details = await self.client.place_details(
            place_id=place_id,
            fields=[
                "name", "formatted_address", "geometry",
                "types", "rating", "user_ratings_total",
                "reviews", "photos", "website", "opening_hours",
                "price_level", "editorial_summary",
            ]
        )

        return self._transform_to_entity_details(details)
```

9.2.3 Additional Data Sources

Beyond Google Maps, the pipeline supports:

- **Instagram**: Visual content, aesthetic signals, engagement metrics
- **Web Scraping**: Entity websites for detailed descriptions and imagery
- **CSV/JSON Import**: Bulk import from curated datasets
- **API Integrations**: Domain-specific data providers

9.3 AI-Powered Enrichment

Raw data is enriched through multiple AI analyzers that extract structured insights.

9.3.1 Vision Analysis

GPT-4 Vision analyzes entity images to extract aesthetic attributes:

Listing 9.3: Vision analyzer implementation

```
class VisionAnalyzer:
    """Analyze images using GPT-4 Vision."""

    async def analyze_entity_images(
        self,
        images: List[bytes],
        entity_type: str
    ) -> VisionAnalysis:
        """Extract visual attributes from entity images."""

        prompt = self.prompts.get("vision_analysis").format(
            entity_type=entity_type,
            dimensions=self.config.vibe_dimensions
        )

        # Prepare image inputs
        image_contents = [
            {"type": "image_url", "image_url": {"url":
               self._to_data_url(img)}}
            for img in images[:self.config.max_images]
        ]

        response = await self.llm.ainvoke(
            model="gpt-4o",
            messages=[{
                "role": "user",
                "content": [{"type": "text", "text": prompt}] +
                   image_contents
            }]
        )

        return self._parse_vision_response(response)
```

The vision analyzer extracts:

- Atmosphere assessment (formal/casual, busy/intimate)
- Design style classification

- Quality signals (craftsmanship visible, attention to detail)
- Category confirmation or refinement

9.3.2 Review Sentiment Analysis

The Vibe Extractor processes reviews to understand entity characteristics:

Listing 9.4: Vibe extraction from reviews

```python
class VibeExtractor:
    """Extract vibe dimensions from review text."""

    async def extract_vibe_dna(
        self,
        reviews: List[Review],
        entity_type: str
    ) -> VibeDNA:
        """Compute Vibe DNA from review corpus."""

        # Aggregate review texts
        review_text = self._prepare_review_corpus(reviews)

        prompt = self.prompts.get("vibe_extraction").format(
            entity_type=entity_type,
            dimensions=self._format_dimensions(),
            reviews=review_text
        )

        response = await self.llm.ainvoke(
            model="gpt-4o",
            messages=[{"role": "user", "content": prompt}]
        )

        # Parse structured scores
        scores = self._parse_vibe_scores(response)

        return VibeDNA(
            scores=scores,
            confidence=self._compute_confidence(len(reviews)),
            extracted_at=datetime.utcnow()
        )
```

9.3.3 Embedding Generation

Entity text is embedded for semantic search:

Listing 9.5: Embedding generation

```python
class EmbeddingGenerator:
    """Generate embeddings for entity content."""

    async def embed_entity(
        self,
        entity: EntityDetails
    ) -> EntityEmbedding:
        """Generate searchable embedding for entity."""

        # Construct embedding text
        text = self._build_embedding_text(
            name=entity.name,
            description=entity.description,
            category=entity.category,
            vibe_summary=entity.vibe_dna.summary,
            reviews_summary=entity.reviews_summary
        )

        # Generate dense embedding
        dense = await self.embedder.embed(text)

        # Generate sparse embedding for hybrid search
        sparse = self.sparse_encoder.encode(text)

        return EntityEmbedding(
            dense=dense,
            sparse=sparse,
            text_hash=self._hash_text(text)
        )
```

9.4 Enrichment Orchestrator

The Master Ingestor orchestrates the complete enrichment flow:

Listing 9.6: Enrichment orchestration

```python
class EnrichmentOrchestrator:
    """Orchestrate entity enrichment pipeline."""

    async def enrich_entity(
        self,
        details: EntityDetails
    ) -> EnrichedEntity:
        """Apply all enrichment steps to an entity."""

        # Parallel enrichment tasks
        async with asyncio.TaskGroup() as tg:
            vision_task = tg.create_task(
                self.vision.analyze_entity_images(
                    details.images,
                    details.entity_type
                )
            )
            vibe_task = tg.create_task(
                self.vibe.extract_vibe_dna(
                    details.reviews,
                    details.entity_type
                )
            )
            embedding_task = tg.create_task(
                self.embedder.embed_entity(details)
            )

        # Merge enrichments
        return EnrichedEntity(
            **details.dict(),
            vision_analysis=vision_task.result(),
            vibe_dna=vibe_task.result(),
            embedding=embedding_task.result()
        )
```

9.5 Temporal Workflows

Temporal.io provides durable workflow execution for the ingestion pipeline.

9.5.1 Workflow Definition

Listing 9.7: Temporal ingestion workflow

```python
from temporalio import workflow
from temporalio.common import RetryPolicy

@workflow.defn
class IngestionWorkflow:
    """Durable workflow for entity ingestion."""

    @workflow.run
    async def run(self, params: IngestionParams) -> IngestionResult:
        """Execute complete ingestion pipeline."""

        # Step 1: Discover entities
        discovered = await workflow.execute_activity(
            discover_entities,
            params.search_params,
            start_to_close_timeout=timedelta(minutes=30),
            retry_policy=RetryPolicy(maximum_attempts=3)
        )

        # Step 2: Process each entity
        results = []
        for entity in discovered:
            try:
                # Extract details
                details = await workflow.execute_activity(
                    extract_entity_details,
                    entity,
                    start_to_close_timeout=timedelta(minutes=5),
                )

                # Enrich with AI
                enriched = await workflow.execute_activity(
                    enrich_entity,
                    details,
                    start_to_close_timeout=timedelta(minutes=10),
                )

                # Write to databases
                await workflow.execute_activity(
                    write_to_trinity,
                    enriched,
                    start_to_close_timeout=timedelta(minutes=2),
                )

                results.append(IngestionSuccess(entity.external_id))

            except Exception as e:
                results.append(IngestionFailure(entity.external_id, str(e)))

        return IngestionResult(
            total=len(discovered),
            succeeded=len([r for r in results if r.success]),
            failed=len([r for r in results if not r.success]),
            details=results
        )
```

9.5.2 Activity Implementations

Activities execute individual pipeline steps with independent retry policies:

Listing 9.8: Temporal activities

```
from temporalio import activity

@activity.defn
async def discover_entities(params: SearchParams) -> List[DiscoveredEntity]:
    """Activity: Discover entities from sources."""
    scraper = get_scraper(params.source)
    entities = []
    async for entity in scraper.discover(params):
        entities.append(entity)
    return entities

@activity.defn
async def enrich_entity(details: EntityDetails) ->
```

```
   EnrichedEntity:
    """Activity: Apply AI enrichment."""
    orchestrator = get_enrichment_orchestrator()
    return await orchestrator.enrich_entity(details)

@activity.defn
async def write_to_trinity(entity: EnrichedEntity) -> None:
    """Activity: Write to all three databases."""
    writer = get_trinity_writer()
    await writer.write_entity(entity)
```

9.6 Change Data Capture

For entities with external sources that update independently, CDC maintains synchronization.

9.6.1 Debezium Integration

Listing 9.9: CDC configuration

```
# config/cdc_config.yaml
connectors:
  source_database:
    connector: postgresql
    database: source_db
    tables:
      - entities
      - entity_attributes
    transforms:
      - route_by_type

  sink_arbor:
    connector: jdbc
    target: arbor_postgres
    mode: upsert
    pk_mode: record_key
```

9.6.2 CDC Processing Pipeline

Listing 9.10: CDC event processing

```python
async def process_cdc_event(event: CDCEvent) -> None:
    """Process a change data capture event."""

    match event.operation:
        case "INSERT":
            # Queue new entity for ingestion
            await ingestion_queue.enqueue(
                IngestionTask(
                    external_id=event.after["external_id"],
                    source=event.source,
                    priority="normal"
                )
            )

        case "UPDATE":
            # Re-enrich if significant fields changed
            if has_significant_changes(event.before, event.after):
                await re_enrichment_queue.enqueue(
                    entity_id=event.after["id"]
                )

        case "DELETE":
            # Mark entity as inactive
            await mark_entity_inactive(event.before["id"])
```

9.7 Quality Control

Ingestion quality is maintained through multiple mechanisms.

9.7.1 Validation Rules

Each entity type has domain-specific validation rules:

Listing 9.11: Entity validation

```python
class EntityValidator:
    """Validate ingested entities."""

    async def validate(
        self,
        entity: EnrichedEntity
    ) -> ValidationResult:
        """Apply all validation rules."""

        errors = []
        warnings = []

        # Required fields
        if not entity.name:
            errors.append("Missing required field: name")

        # Vibe DNA completeness
        missing_dims =
           self._check_vibe_completeness(entity.vibe_dna)
        if missing_dims:
            warnings.append(f"Incomplete Vibe DNA:
               {missing_dims}")

        # Embedding quality
        if entity.embedding.confidence < 0.7:
            warnings.append("Low embedding confidence")

        # Domain-specific rules
        errors.extend(
            await self.domain_validator.validate(entity)
        )

        return ValidationResult(
            valid=len(errors) == 0,
            errors=errors,
            warnings=warnings
        )
```

9.7.2 Duplicate Detection

Before writing, duplicates are identified and resolved:

Listing 9.12: Duplicate detection

```python
async def detect_duplicates(
    entity: EnrichedEntity
) -> Optional[str]:
    """Check for existing duplicate entities."""

    # Check by external ID
    existing = await repo.get_by_external_id(entity.external_id)
    if existing:
        return existing.id

    # Check by name + location similarity
    candidates = await repo.search_nearby(
        name=entity.name,
        location=entity.location,
        radius_meters=100
    )

    for candidate in candidates:
        similarity = compute_name_similarity(entity.name, candidate.name)
        if similarity > 0.85:
            return candidate.id

    return None
```

9.8 Monitoring and Alerting

The ingestion pipeline is closely monitored to ensure continuous operation.

9.8.1 Key Metrics

- **Throughput**: Entities processed per hour
- **Latency**: Time from discovery to availability
- **Success Rate**: Percentage of entities successfully ingested

- **Enrichment Quality**: Vibe DNA completeness, embedding confidence
- **Queue Depth**: Pending items in each processing stage

9.8.2 Alerting Conditions

Alerts fire when:

- Success rate drops below 95%
- Queue depth exceeds threshold for extended period
- Enrichment API error rate spikes
- Workflow failures exceed normal baseline

Chapter 10

Discovery Engine

The Discovery Engine is the heart of ARBOR's user-facing functionality, orchestrating the transformation of natural language queries into ranked, explained recommendations. This chapter examines the end-to-end discovery flow, from query understanding to response delivery.

10.1 Query Processing

Discovery begins when a user submits a query expressing their needs.

10.1.1 Query Reception

The API endpoint receives and validates incoming queries:

Listing 10.1: Discovery endpoint implementation

```
@router.post("/discover", response_model=DiscoveryResponse)
async def discover(
    request: DiscoveryRequest,
    user: Optional[User] = Depends(get_current_user),
    session: AsyncSession = Depends(get_session)
) -> DiscoveryResponse:
    """Main discovery endpoint."""

    # Build initial state
    state = AgentState(
        user_query=request.query,
        user_id=user.id if user else None,
        user_location=request.location,
        conversation_history=await get_conversation_history(
```

```python
            request.session_id
        ),
        trace_id=generate_trace_id(),
    )

    # Execute agent graph
    result = await discovery_graph.ainvoke(state)

    # Emit analytics event
    await emit_search_event(request, result)

    return DiscoveryResponse(
        message=result["final_response"],
        recommendations=result["recommendations"],
        trace_id=state["trace_id"]
    )
```

10.1.2 Context Assembly

Each query is enriched with contextual information:

- **User Profile**: Preferences, history, and saved entities if authenticated
- **Conversation History**: Previous turns in ongoing conversations
- **Location Context**: User's current or specified location
- **Temporal Context**: Time of day, day of week, seasonality

10.2 Intent Understanding

The Intent Router classifies queries and extracts actionable parameters.

10.2.1 Classification Process

Query classification determines how the system processes the request:

Listing 10.2: Intent classification prompt

```
# config/prompts/intent_classification.txt

You are analyzing a query for a {domain_name} discovery system.
```

```

Query: {query}

Classify the intent and extract relevant parameters.

Intent categories:
- RECOMMENDATION: User wants suggestions matching criteria
- COMPARISON: User wants to compare specific entities
- EXPLORATION: User is browsing without specific criteria
- DETAIL: User wants information about a known entity
- CLARIFICATION: User is refining a previous query
- OUT_OF_SCOPE: Query is outside system capabilities

Available filters:
{filter_schema}

Respond in JSON format:
{
    "intent": "<category>",
    "confidence": <0-1>,
    "filters": {...},
    "entities_mentioned": [...],
    "vibe_preferences": {...}
}
```

10.2.2 Filter Extraction

Structured filters are extracted from natural language:

Query Fragment	Extracted Filter	Type
“in Milan”	`city: "Milano"`	Location
“elegant but casual”	`formality: 40-60`	Vibe range
“under 100 euros”	`price_tier: [1, 2]`	Price
“Neapolitan style”	`style: "neapolitan"`	Category
“near Duomo”	`near: {lat, lng}`	Proximity

Table 10.1: Examples of filter extraction

10.3 Multi-Source Retrieval

With intent classified and filters extracted, retrieval agents gather relevant entities.

10.3.1 Parallel Execution

Independent retrieval operations execute concurrently to minimize latency:

Listing 10.3: Parallel retrieval orchestration

```python
async def parallel_retrieval(state: AgentState) -> AgentState:
    """Execute all retrieval agents in parallel."""

    async with asyncio.TaskGroup() as tg:
        # Semantic search
        vector_task = tg.create_task(
            vector_agent.search(state)
        )

        # Structured filtering
        metadata_task = tg.create_task(
            metadata_agent.query(state)
        )

    # Merge results
    state["vector_results"] =
       vector_task.result()["vector_results"]
    state["metadata_results"] =
       metadata_task.result()["metadata_results"]

    # Graph exploration depends on initial results
    state = await historian_agent.explore(state)

    return state
```

10.3.2 Retrieval Strategies

Different query types emphasize different retrieval sources:

Query Type	Strategy
Semantic (“somewhere cozy”)	Vector search primary, metadata secondary
Filtered (“tailors in Rome”)	Metadata primary, vector for ranking
Relationship (“similar to X”)	Graph primary with vector similarity
Hybrid	Equal weight, RRF fusion

Table 10.2: Retrieval strategies by query type

10.4 Result Fusion and Ranking

Results from multiple sources are merged and ranked before synthesis.

10.4.1 Reciprocal Rank Fusion

RRF combines ranked lists from multiple sources:

Listing 10.4: RRF implementation

```python
def reciprocal_rank_fusion(
    ranked_lists: List[List[EntityResult]],
    k: int = 60
) -> List[EntityResult]:
    """Combine ranked lists using RRF."""

    scores = defaultdict(float)
    entities = {}

    for ranked_list in ranked_lists:
        for rank, entity in enumerate(ranked_list, start=1):
            scores[entity.id] += 1 / (k + rank)
            entities[entity.id] = entity

    # Sort by fused score
    sorted_ids = sorted(
        scores.keys(),
        key=lambda x: scores[x],
        reverse=True
    )

    return [entities[id] for id in sorted_ids]
```

10.4.2 ML Reranking

After fusion, ML models refine the ranking:

Listing 10.5: Reranking pipeline

```python
class RerankerPipeline:
    """Multi-stage reranking pipeline."""

    async def rerank(
        self,
        query: str,
        candidates: List[EntityResult],
        limit: int = 10
    ) -> List[EntityResult]:
        """Apply reranking stages."""

        # Stage 1: Cohere rerank for relevance
        if len(candidates) > 50:
            candidates = await self.cohere_rerank(
                query, candidates, top_k=50
            )

        # Stage 2: Vibe alignment scoring
        candidates = self.score_vibe_alignment(
            query, candidates
        )

        # Stage 3: Diversity injection
        candidates = self.inject_diversity(candidates)

        # Stage 4: Final cutoff
        return candidates[:limit]
```

10.5 Personalization

For authenticated users, results are personalized based on history and preferences.

10.5.1 User Profile Integration

Listing 10.6: Personalization layer

```python
class PersonalizationLayer:
    """Apply user-specific personalization."""

    async def personalize(
        self,
        user_id: str,
        candidates: List[EntityResult]
    ) -> List[EntityResult]:
        """Adjust ranking based on user profile."""

        profile = await self.get_user_profile(user_id)

        for candidate in candidates:
            # Boost entities matching user preferences
            pref_score = self.compute_preference_alignment(
                candidate.vibe_dna,
                profile.vibe_preferences
            )

            # Penalize already-seen entities
            if candidate.id in profile.viewed_entities:
                candidate.score *= 0.7

            # Boost based on similar liked entities
            similarity_boost = self.compute_similarity_to_liked(
                candidate,
                profile.liked_entities
            )

            candidate.score += pref_score + similarity_boost

        return sorted(candidates, key=lambda x: x.score,
            reverse=True)
```

10.5.2 Preference Learning

User preferences are learned from implicit and explicit signals:

- **Clicks**: Entities the user explored in detail

- **Saves**: Explicitly saved entities
- **Conversion**: Entities where the user took action
- **Dwell Time**: Time spent viewing entity details
- **Explicit Ratings**: Direct feedback when provided

10.6 Response Synthesis

The Curator agent transforms ranked results into coherent recommendations.

10.6.1 Synthesis Strategy

Response synthesis follows a structured approach:

1. **Context Framing**: Acknowledge the user's request and any constraints
2. **Primary Recommendations**: Present top matches with explanations
3. **Relationship Insights**: Note relevant connections between entities
4. **Alternatives**: Suggest alternatives if main recommendations have caveats
5. **Follow-up Invitation**: Offer to refine or explore further

10.6.2 Explanation Generation

Each recommendation includes substantive justification:

Listing 10.7: Example recommendation with explanation

```
**Sartoria Ciardi** emerges as a strong match for your
requirements. The workshop maintains the Neapolitan
construction you -specifiedunstructured shoulders,
hand-stitched -buttonholeswhile offering a more
accessible price point than some competitors.

What distinguishes Ciardi is the direct lineage:
Maestro Ciardi trained under Antonio Panico,
widely considered the dean of Neapolitan tailoring.
The atmosphere is intimate (they take perhaps three
clients at a time), which suits your preference for
a personal experience over a high-volume operation.
```

10.7 Conversation Management

Discovery extends beyond single queries to multi-turn conversations.

10.7.1 Context Persistence

Conversation state persists across turns:

Listing 10.8: Conversation state management

```python
class ConversationStore:
    """Persist and retrieve conversation context."""

    async def get_context(
        self,
        session_id: str
    ) -> ConversationContext:
        """Retrieve conversation context."""

        data = await self.redis.get(f"conv:{session_id}")
        if not data:
            return ConversationContext.new()

        return ConversationContext.parse_raw(data)

    async def update_context(
        self,
        session_id: str,
        user_message: str,
        assistant_response: str,
        recommendations: List[Recommendation]
    ) -> None:
        """Update conversation with new turn."""

        context = await self.get_context(session_id)

        context.turns.append(ConversationTurn(
            user=user_message,
            assistant=assistant_response,
            recommendations=recommendations,
            timestamp=datetime.utcnow()
        ))

        # Retain last N turns to manage context size
        context.turns = context.turns[-self.max_turns:]

        await self.redis.set(
            f"conv:{session_id}",
            context.json(),
            ex=self.ttl_seconds
        )
```

10.7.2 Reference Resolution

Follow-up queries reference previous context:

Listing 10.9: Reference resolution

```
def resolve_references(
    query: str,
    context: ConversationContext
) -> str:
    """Resolve pronouns and references to previous turns."""

    # Expand "the first one" → entity name
    if "first one" in query.lower() or "first option" in query.lower():
        if context.last_recommendations:
            entity_name = context.last_recommendations[0].name
            query = query.replace("the first one", entity_name)

    # Expand "that one" / "it" based on recency
    if "that" in query.lower() or query.lower().startswith("it"):
        if context.last_discussed_entity:
            query = query.replace("that", context.last_discussed_entity)

    # Expand "similar" → similar to last recommendation
    if "similar" in query.lower() and "to" not in query.lower():
        if context.last_recommendations:
            query = f"{query} to {context.last_recommendations[0].name}"

    return query
```

10.8 Performance Considerations

Meeting latency targets requires optimization at every stage.

10.8.1 Latency Budget

The 2.5-second P95 target is allocated across stages:

Stage	**Target (ms)**	**Max (ms)**
Request parsing	10	20
Intent classification	200	400
Parallel retrieval	150	300
Graph exploration	100	200
Fusion & reranking	50	100
Curator synthesis	800	1200
Response formatting	10	20
Total	1320	2240

Table 10.3: Latency budget allocation

10.8.2 Optimization Techniques

- **Streaming**: Begin response delivery before synthesis completes
- **Precomputation**: Cache common query patterns
- **Early Termination**: Stop retrieval when sufficient candidates found
- **Result Size Limits**: Cap candidates at each stage

10.9 Error Recovery

The discovery engine handles failures gracefully.

10.9.1 Fallback Strategies

When components fail, fallbacks preserve functionality:

Listing 10.10: Discovery fallback handling

```python
async def discover_with_fallbacks(state: AgentState) -> AgentState:
    """Execute discovery with fallback handling."""

    try:
        # Normal flow
        state = await intent_router.analyze(state)
        state = await parallel_retrieval(state)
        state = await curator.synthesize(state)

    except VectorSearchError:
        # Fallback to metadata-only search
        logger.warning("Vector search failed, using metadata fallback")
        state = await metadata_only_search(state)
        state = await curator.synthesize(state)

    except LLMError:
        # Fallback to template-based response
        logger.error("LLM synthesis failed, using template")
        state = await template_synthesis(state)

    except Exception as e:
        # Generic fallback
        logger.exception("Discovery failed")
        state["final_response"] = (
            "I apologize, but I'm having trouble processing your "
            "request right now. Please try again in a moment."
        )

    return state
```

10.9.2 Partial Result Handling

When some retrieval sources fail, others continue:

- If Qdrant fails: Use PostgreSQL full-text search
- If Neo4j fails: Omit relationship context from response
- If reranker fails: Use fusion scores directly

Chapter 11

Machine Learning Pipeline

ARBOR's machine learning infrastructure incorporates 24 distinct modules addressing challenges from knowledge graph reasoning to explainability. This chapter surveys these capabilities.

11.1 ML Architecture Overview

The ML pipeline serves multiple objectives:

- **Quality Enhancement**: Improving ranking relevance
- **Cost Optimization**: Reducing inference costs
- **Continuous Learning**: Adapting to feedback
- **Operational Intelligence**: Detecting drift and anomalies

11.2 Knowledge Graph Reasoning

The `kg_reasoning.py` module implements graph neural network techniques for entity resolution and relationship prediction.

11.2.1 Entity Resolution

When ingested data may refer to the same entity under different identifiers, graph reasoning resolves these using neighbor similarity, name matching, and type compatibility.

11.2.2 Link Prediction

The system predicts relationships: training lineages between artisans, brand-retailer associations, style affiliations, and competitive relationships.

11.3 Reranking Pipeline

The `reranking_pipeline.py` module implements multi-stage ranking refinement:

1. **Semantic Reranking**: Cross-encoder for relevance
2. **Vibe Alignment**: Score entities on preference match
3. **Freshness Weighting**: Boost recent entities
4. **Diversity Injection**: Ensure variety
5. **Business Rules**: Domain-specific adjustments

11.4 Cost-Aware Routing

The `cost_aware_router.py` module optimizes LLM selection:

- Estimates query complexity
- Tracks model costs and latencies
- Selects cheapest model meeting quality threshold
- Respects latency constraints

11.5 Feature Store

The `feature_store.py` module provides real-time feature serving:

Category	Examples
Entity Features	Vibe DNA scores, category, price tier
User Features	Preference vectors, session count
Query Features	Length, complexity, filter count
Interaction Features	Query-entity similarity, CTR

Table 11.1: Feature categories

11.6 Explainability

The `explainability.py` module generates interpretable explanations:

SHAP Values Feature attribution for rankings

LIME Local explanations for recommendations

Attention Query-attribute matching

Counterfactual What would change this recommendation

11.7 Drift Detection

The `drift_detection.py` module monitors distribution shifts:

- Data drift in entity attributes
- Concept drift in preferences
- Model degradation
- Embedding space movement

11.8 Additional ML Modules

`prompt_optimizer.py` Automatic prompt tuning

`causal_inference.py` Treatment effect estimation

`federated_learning.py` Privacy-preserving training

`rlhf.py` Human feedback integration

`ab_testing.py` Experimentation framework

`predictive_prefetch.py` Query anticipation

`knowledge_distillation.py` Model compression

`adversarial_testing.py` Robustness testing

`synthetic_data.py` Training data generation

`graph_expansion.py` Relationship discovery

`personalization.py` User preference modeling

`rag_evaluation.py` Retrieval quality metrics

Each module follows ARBOR's patterns for configuration, observability, and testing.

Chapter 12

Event-Driven Architecture

ARBOR employs event-driven patterns to enable loose coupling, real-time analytics, and asynchronous processing. Apache Kafka serves as the event backbone, with producers and consumers distributed across the system.

12.1 Event-Driven Design

12.1.1 Design Rationale

Event-driven architecture provides several benefits:

Decoupling Components communicate through events rather than direct calls

Scalability Consumers scale independently based on throughput needs

Durability Events are persisted for replay and recovery

Analytics Event streams enable real-time and historical analysis

12.2 Kafka Topology

12.2.1 Topic Structure

Events flow through organized topics:

Topic	Contents
`entity.ingested`	New entities entering the system
`entity.updated`	Entity attribute changes
`query.executed`	Discovery queries and results
`user.feedback`	Explicit and implicit feedback signals
`ml.predictions`	Model inference results
`system.events`	Operational events and metrics

Table 12.1: Primary Kafka topics

12.2.2 Event Schema

All events follow a consistent envelope:

Listing 12.1: Event envelope schema

```python
class EventEnvelope(BaseModel):
    event_id: str = Field(default_factory=lambda: str(uuid4()))
    event_type: str
    timestamp: datetime = Field(default_factory=datetime.utcnow)
    source: str
    trace_id: Optional[str]
    payload: Dict[str, Any]
```

12.3 Event Producers

12.3.1 Query Events

Every discovery query emits events for analytics:

Listing 12.2: Query event production

```python
class QueryEventProducer:
    async def emit_query_event(
        self,
        query: str,
        results: List[EntityResult],
        latency_ms: int,
        trace_id: str
    ) -> None:
        event = EventEnvelope(
            event_type="query.executed",
            source="discovery_engine",
            trace_id=trace_id,
            payload={
                "query": query,
                "result_count": len(results),
                "result_ids": [r.id for r in results[:10]],
                "latency_ms": latency_ms,
            }
        )
        await self.producer.send("query.executed", event.json())
```

12.3.2 Entity Change Events

Entity lifecycle changes propagate through events enabling cache invalidation and search index updates.

12.4 Event Consumers

12.4.1 Analytics Pipeline

Query events feed the analytics pipeline:

- Query pattern analysis
- Popular entity tracking
- Latency monitoring
- Conversion funnel analysis

12.4.2 ML Feedback Loop

User feedback events trigger model updates:

Listing 12.3: Feedback consumer

```
class FeedbackConsumer:
    async def process(self, event: EventEnvelope) -> None:
        feedback = FeedbackSignal(**event.payload)
```

```python

        # Update feature store
        await self.feature_store.record_interaction(
            user_id=feedback.user_id,
            entity_id=feedback.entity_id,
            signal=feedback.signal
        )

        # Check if retraining threshold reached
        if await self.should_retrain():
            await self.training_queue.enqueue(RetrainJob())
```

12.5 Cache Invalidation

Entity changes trigger cache invalidation across Redis and CDN:

Listing 12.4: Cache invalidation consumer

```python
class CacheInvalidator:
    async def handle_entity_update(
        self,
        event: EventEnvelope
    ) -> None:
        entity_id = event.payload["entity_id"]

        # Invalidate Redis caches
        await self.redis.delete(f"entity:{entity_id}")
        await self.redis.delete(f"features:{entity_id}")

        # Purge CDN cache
        await self.cdn.purge(f"/api/entities/{entity_id}")
```

12.6 Stream Processing

For real-time aggregations, stream processing augments batch analytics:

- Rolling query counts by category
- Real-time trending entities

- Active user session tracking
- Anomaly detection on query patterns

12.7 Operational Considerations

12.7.1 Retention Policies

- Query events: 30 days retention
- Feedback events: 90 days retention
- Entity events: Compacted (keep latest)

12.7.2 Consumer Groups

Each processing concern uses isolated consumer groups, enabling independent scaling and failure isolation.

Part IV

Frontend & User Experience

Chapter 13

Frontend Architecture

ARBOR's frontend provides the primary interface for discovery interactions, curator administration, and system management. Built on Next.js with TypeScript, the frontend emphasizes performance, accessibility, and beautiful design.

13.1 Technology Stack

13.1.1 Core Technologies

Technology	Purpose
Next.js 14	React framework with SSR/SSG
TypeScript	Type safety and developer experience
shadcn/ui	Component library
Tailwind CSS	Utility-first styling
TanStack Query	Server state management
Zustand	Client state management

Table 13.1: Frontend technology stack

13.1.2 Selection Rationale

Next.js Server components reduce client bundle, improve SEO, and enable streaming

TypeScript Catches errors early, improves refactoring confidence

shadcn/ui Unstyled primitives enable design customization

TanStack Query Handles caching, background refresh, optimistic updates

13.2 Application Structure

13.2.1 Directory Organization

frontend/ |– src/ | |– app/ Next.js App Router | | |– (discover)/ Discovery routes | | |– (admin)/ Admin routes | | ‘– api/ API routes | |– components/ | | |– ui/ shadcn primitives | | |– discovery/ Discovery-specific | | ‘– admin/ Admin-specific | |– hooks/ Custom React hooks | |– lib/ Utilities | ‘– types/ TypeScript types |– public/ Static assets ‘– tests/ Test files

13.2.2 Routing Strategy

The App Router enables route groups for logical separation:

- **(discover)**: Public discovery interface
- **(admin)**: Authenticated curator/admin area
- **api**: Backend-for-frontend endpoints

13.3 Data Fetching

13.3.1 Server Components

Data fetching occurs on the server where possible:

Listing 13.1: Server component data fetching

```
// app/(discover)/entity/[id]/page.tsx
async function EntityPage({ params }: { params: { id: string } }) {
  const entity = await fetchEntity(params.id);
  const related = await fetchRelatedEntities(params.id);

  return (
    <EntityDetail entity={entity}>
      <RelatedEntities entities={related} />
    </EntityDetail>
  );
}
```

13.3.2 Client-Side Queries

Interactive features use TanStack Query:

Listing 13.2: Client-side data fetching

```
function useDiscoveryQuery(query: string) {
  return useQuery({
    queryKey: ['discover', query],
    queryFn: () => discoverApi.search(query),
    staleTime: 1000 * 60 * 5, // 5 minutes
    gcTime: 1000 * 60 * 30,   // 30 minutes
  });
}
```

13.4 State Management

13.4.1 Client State with Zustand

UI state uses Zustand for simplicity:

Listing 13.3: Zustand store

```
const useDiscoveryStore = create<DiscoveryState>((set) => ({
  query: '',
  filters: {},
  setQuery: (query) => set({ query }),
  setFilter: (key, value) =>
    set((state) => ({
      filters: { ...state.filters, [key]: value }
    })),
  clearFilters: () => set({ filters: {} }),
}));
```

13.4.2 Server State with TanStack Query

Server state (entities, recommendations) is managed separately from UI state, enabling proper cache invalidation and background updates.

13.5 Performance Optimization

13.5.1 Streaming SSR

Long-running data fetches use streaming:

Listing 13.4: Streaming with Suspense

```
function DiscoveryPage() {
  return (
    <div>
      <SearchInput />
      <Suspense fallback={<ResultsSkeleton />}>
        <DiscoveryResults />
      </Suspense>
    </div>
  );
}
```

13.5.2 Image Optimization

- Next.js Image component for automatic optimization
- Cloudflare Image Resizing at the edge
- WebP format with fallbacks
- Lazy loading with blur placeholders

13.6 Design System

13.6.1 Token-Based Design

CSS custom properties define the design system:

Listing 13.5: Design tokens

```
:root {
  --color-primary: oklch(0.7 0.15 250);
  --color-surface: oklch(0.98 0.01 250);
  --radius-md: 0.5rem;
  --spacing-4: 1rem;
  --font-sans: 'Inter', system-ui, sans-serif;
}
```

13.6.2 Component Variants

Components support variants for flexibility:

Listing 13.6: Button variants

```
const buttonVariants = cva(
  "inline-flex items-center justify-center rounded-md",
  {
    variants: {
      variant: {
        primary: "bg-primary text-white hover:bg-primary/90",
        secondary: "bg-secondary text-secondary-foreground",
        ghost: "hover:bg-accent hover:text-accent-foreground",
      },
      size: {
        sm: "h-8 px-3 text-sm",
        md: "h-10 px-4",
        lg: "h-12 px-6 text-lg",
      },
    },
    defaultVariants: {
      variant: "primary",
      size: "md",
    },
  }
);
```

13.7 Accessibility

13.7.1 Standards Compliance

The frontend targets WCAG 2.1 AA compliance:

- Semantic HTML structure
- Keyboard navigation support

- Screen reader compatibility
- Color contrast requirements
- Focus management

13.7.2 Testing

Accessibility is validated through:

- Automated axe-core testing in CI
- Manual testing with screen readers
- Keyboard-only navigation testing

13.8 Mobile Responsiveness

13.8.1 Responsive Strategy

The design adapts across breakpoints:

- Mobile-first CSS approach
- Touch-optimized interactions
- Adaptive layout components
- Progressive enhancement

Chapter 14

UI Components

The ARBOR interface comprises carefully designed components that balance aesthetic appeal with functional clarity. This chapter examines the key UI components that define the user experience.

14.1 Discovery Components

14.1.1 Conversational Search

The primary discovery interface presents as a conversational chat:

SearchInput Natural language query input with suggestions

MessageList Conversation history display

RecommendationCard Rich entity presentation

LoadingStates Skeleton and streaming indicators

14.1.2 Entity Cards

Entity cards present discoveries with visual richness:

- Hero image with lazy loading
- Name and category badges
- Vibe DNA visualization (radar chart)
- Quick actions (save, share, details)
- Curator explanation excerpt

14.1.3 Vibe Visualizations

The Vibe DNA is visualized through:

- Radar charts for multi-dimensional display
- Animated transitions between states
- Color-coded dimension axes
- Comparative overlays for entity comparison

14.2 Filter Components

14.2.1 Faceted Filters

Structured filtering complements natural language:

CategoryFilter Hierarchical category selection

LocationFilter Map-based or address search

PriceFilter Tier selection with visual indicators

VibeSliders Dimension preference adjustment

14.2.2 Active Filters

Applied filters display as dismissible chips, enabling easy modification.

14.3 Map Components

For location-aware discovery:

EntityMap Interactive map with entity markers

ClusterMarkers Grouped markers at zoom levels

MapFilters Geographic radius selection

EntityPopup Quick entity preview on marker click

14.4 Detail Components

Entity detail views provide comprehensive information:

EntityHeader Hero image, name, key attributes

VibeProfile Full dimension breakdown

RelationshipGraph Visual relationship exploration

ReviewSummary AI-synthesized review insights

ActionBar Contact, directions, save actions

14.5 Admin Components

The curator dashboard includes specialized components:

EntityTable Sortable, filterable entity list

ValidationPanel Approve/reject interface

EnrichmentStatus Processing pipeline status

VibeEditor Manual Vibe DNA adjustment

RelationshipEditor Graph relationship management

14.6 Feedback Components

User feedback collection:

RatingWidget Simple thumbs up/down

FeedbackForm Detailed feedback collection

ReportDialog Issue reporting interface

14.7 Animation and Motion

14.7.1 Motion Principles

Animations follow principles of purposeful motion:

- Subtle transitions (150-300ms)
- Meaningful direction (enter from action source)
- Reduced motion respect (prefers-reduced-motion)
- Performance-first (GPU-accelerated transforms)

14.7.2 Implementation

Animations use Framer Motion:

Listing 14.1: Animation example

```
const cardVariants = {
  hidden: { opacity: 0, y: 20 },
  visible: {
    opacity: 1,
    y: 0,
    transition: { duration: 0.3, ease: "easeOut" }
  },
  exit: { opacity: 0, scale: 0.95 }
};
```

14.8 Component Testing

14.8.1 Testing Strategy

Components are tested at multiple levels:

- Unit tests for logic-heavy components
- Integration tests for composed behaviors
- Visual regression tests with Storybook
- Accessibility tests with axe-core

14.8.2 Storybook Documentation

Each component is documented in Storybook with:

- Interactive variants and states
- Props documentation
- Usage examples
- Accessibility annotations

Chapter 15

Admin Dashboard

The Admin Dashboard provides curators and administrators with tools to manage entities, monitor system health, and maintain knowledge base quality. This chapter examines the dashboard's capabilities.

15.1 Dashboard Overview

15.1.1 User Roles

The dashboard supports multiple roles:

Role	**Capabilities**
Curator	Validate entities, edit Vibe DNA, manage relationships
Domain Admin	Configure domain settings, manage curators
System Admin	Full access, user management, system config

Table 15.1: Admin dashboard roles

15.1.2 Navigation Structure

- **Entities**: Browse, search, validate entities
- **Queue**: Pending validation items
- **Analytics**: Usage and quality metrics
- **Configuration**: Domain and system settings

15.2 Entity Management

15.2.1 Entity Browser

The entity browser provides:

- Searchable, sortable entity table
- Quick filters by status, category, validation level
- Bulk actions for efficient processing
- Export capabilities

15.2.2 Entity Editor

Detailed entity editing includes:

- Core attribute editing
- Vibe DNA score adjustment
- Image management
- Relationship editing
- Validation history

15.3 Validation Workflow

15.3.1 Validation Queue

New entities enter the validation queue:

1. AI-ingested entities appear with enriched data
2. Curator reviews accuracy and completeness
3. Curator adjusts Vibe scores if needed
4. Entity is approved, rejected, or sent for re-enrichment

15.3.2 Validation Interface

The validation panel presents:

- Entity preview matching public display
- AI enrichment results with confidence scores
- Comparison against similar entities
- Quick approve/reject actions
- Detailed editing access

15.4 Relationship Management

15.4.1 Graph Editor

Visual relationship management:

- Interactive graph visualization
- Add/remove relationship edges
- Relationship type selection from domain schema
- Bulk relationship import

15.4.2 Suggested Relationships

ML-suggested relationships for curator review based on entity similarity and pattern matching.

15.5 Analytics Dashboard

15.5.1 Key Metrics

- Query volume and patterns
- Entity click-through rates
- Validation throughput
- System latency percentiles

15.5.2 Quality Metrics

- User feedback scores
- Vibe DNA coverage completeness
- Relationship density
- Enrichment success rates

15.6 Configuration Management

15.6.1 Domain Configuration

Administrators can manage:

- Vibe dimension definitions
- Category hierarchy
- Relationship types
- Curator persona settings

15.6.2 System Settings

- LLM provider configuration
- Ingestion source settings
- Rate limiting thresholds
- Feature flags

15.7 Audit and Compliance

15.7.1 Activity Logging

All administrative actions are logged:

- Entity modifications with before/after
- Validation decisions

- Configuration changes
- User access events

15.7.2 Audit Trail

Complete audit history enables:

- Change tracking and rollback
- Compliance reporting
- Dispute resolution

Part V

Infrastructure & Operations

Chapter 16

Deployment Architecture

ARBOR's deployment architecture enables reliable, scalable operation across development, staging, and production environments. This chapter examines the containerization, orchestration, and infrastructure-as-code approaches.

16.1 Containerization Strategy

16.1.1 Docker Images

Each service has an optimized container image:

Service	Base Image	Size
Backend API	python:3.12-slim	200MB
Frontend	node:20-alpine	150MB
Worker	python:3.12-slim	250MB

Table 16.1: Docker image configurations

16.1.2 Multi-Stage Builds

Images use multi-stage builds for minimal size:

Listing 16.1: Multi-stage Dockerfile

```
# Build stage
FROM python:3.12-slim AS builder
WORKDIR /app
COPY pyproject.toml poetry.lock ./
RUN pip install poetry && poetry export -o requirements.txt
```

```dockerfile
RUN pip wheel -r requirements.txt -w /wheels

# Runtime stage
FROM python:3.12-slim
COPY --from=builder /wheels /wheels
RUN pip install --no-index /wheels/*
COPY . .
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0"]
```

16.2 Kubernetes Architecture

16.2.1 Cluster Structure

Production uses a multi-node Kubernetes cluster:

- Control plane: 3 nodes for HA
- Application nodes: Auto-scaling pool
- Database nodes: Dedicated node pool with SSDs
- GPU nodes: Optional for local inference

16.2.2 Key Workloads

Deployments Stateless services: API, frontend, workers

StatefulSets Databases: PostgreSQL, Redis, Qdrant

DaemonSets Logging agents, node monitoring

CronJobs Scheduled tasks: cleanup, reports

16.2.3 Service Mesh

Istio provides service mesh capabilities:

- mTLS between services
- Traffic management and canary deployments
- Observability integration
- Circuit breaking

16.3 Infrastructure as Code

16.3.1 Terraform Configuration

Cloud infrastructure is defined in Terraform:

Listing 16.2: Terraform module structure

```
infrastructure/
|-- terraform/
|   |-- modules/
|   |   |-- networking/
|   |   |-- kubernetes/
|   |   |-- databases/
|   |   `-- observability/
|   |-- environments/
|   |   |-- dev/
|   |   |-- staging/
|   |   `-- production/
|   `-- main.tf
```

16.3.2 Environment Parity

Development, staging, and production share configuration with environment-specific variables for sizing and redundancy.

16.4 Database Deployments

16.4.1 PostgreSQL

PostgreSQL uses CloudNativePG operator:

- Primary with synchronous replicas
- Automatic failover
- Continuous WAL archiving to S3
- PgBouncer connection pooling

16.4.2 Qdrant

Qdrant cluster configuration:

- 3-node cluster for redundancy
- Sharding for data distribution
- Snapshot backups to object storage

16.4.3 Neo4j

Neo4j causal cluster:

- 3 core members for write availability
- Read replicas for query scaling
- Online backup to cloud storage

16.5 CI/CD Pipeline

16.5.1 Pipeline Stages

1. **Build**: Compile, test, build images
2. **Security Scan**: Vulnerability scanning
3. **Deploy Staging**: Automatic deployment
4. **Integration Tests**: E2E test suite
5. **Deploy Production**: Manual approval gate
6. **Smoke Tests**: Post-deployment validation

16.5.2 Deployment Strategies

Rolling Default for stateless services

Blue-Green Database schema migrations

Canary New features with gradual rollout

16.6 Scaling Configuration

16.6.1 Horizontal Pod Autoscaling

Listing 16.3: HPA configuration

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api
  minReplicas: 3
  maxReplicas: 20
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

16.6.2 Vertical Pod Autoscaling

VPA recommends and optionally applies resource adjustments based on observed usage patterns.

16.7 Disaster Recovery

16.7.1 Backup Strategy

- Database backups: Continuous + daily snapshots
- Configuration: Version controlled
- Secrets: Encrypted backup

- Recovery testing: Monthly drills

16.7.2 Multi-Region Considerations

For enterprise deployments:

- Active-passive failover region
- Cross-region database replication
- Global load balancing at CDN layer

Chapter 17

Observability

Comprehensive observability enables understanding system behavior, diagnosing issues, and optimizing performance. ARBOR integrates multiple observability tools for tracing, metrics, logs, and LLM-specific monitoring.

17.1 Observability Stack

17.1.1 Core Components

Component	**Purpose**
OpenTelemetry	Unified instrumentation and collection
Langfuse	LLM-specific tracing and analytics
Prometheus	Time-series metrics storage
Grafana	Visualization and dashboards
Loki	Log aggregation and querying

Table 17.1: Observability stack components

17.2 Distributed Tracing

17.2.1 OpenTelemetry Integration

All services emit traces through OpenTelemetry:

Listing 17.1: OpenTelemetry setup

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter
    import (
    OTLPSpanExporter
)

provider = TracerProvider()
provider.add_span_processor(
    BatchSpanProcessor(OTLPSpanExporter())
)
trace.set_tracer_provider(provider)
```

17.2.2 Trace Context

Trace context propagates across:

- HTTP requests (W3C Trace Context)
- Kafka messages (custom headers)
- Database queries (query comments)
- LLM calls (Langfuse trace IDs)

17.3 LLM Observability

17.3.1 Langfuse Integration

Every LLM call is traced through Langfuse:

Listing 17.2: Langfuse tracing

```
from langfuse.decorators import observe

@observe(as_type="generation")
async def generate_synthesis(
    query: str,
    context: str
) -> str:
    response = await llm.ainvoke(
        model="gpt-4o",
        messages=[
            {"role": "system", "content": CURATOR_PROMPT},
            {"role": "user", "content": f"{query}\n\n{context}"}
        ]
    )
    return response.content
```

17.3.2 LLM Metrics

Langfuse captures:

- Token usage (input/output)
- Latency (time to first token, total)
- Cost estimation
- Prompt versions
- User feedback correlation

17.4 Metrics Collection

17.4.1 Application Metrics

Key metrics are collected:

Request Metrics Latency, throughput, error rates

Business Metrics Queries per minute, active users

Resource Metrics CPU, memory, connections

Custom Metrics Cache hit rates, queue depths

17.4.2 SLI/SLO Tracking

Service Level Indicators inform SLOs:

- Discovery latency P95 $<$ 2.5s
- API availability $>$ 99.9%
- Error rate $<$ 0.1%

17.5 Logging Architecture

17.5.1 Structured Logging

All logs are structured JSON:

Listing 17.3: Structured logging

```python
import structlog

logger = structlog.get_logger()

logger.info(
    "query_processed",
    query=query,
    result_count=len(results),
    latency_ms=latency,
    trace_id=trace_id
)
```

17.5.2 Log Aggregation

Logs flow through:

1. Application emits to stdout
2. Fluent Bit collects from containers
3. Loki stores with label indexing
4. Grafana provides querying interface

17.6 Dashboards

17.6.1 Operational Dashboards

System Overview Key health indicators

API Performance Request latency and errors

Database Health Connection pools, query times

LLM Costs Token usage and spend tracking

17.6.2 Business Dashboards

Discovery Analytics Query patterns, popular entities

User Engagement Sessions, conversions

Content Quality Validation rates, feedback

17.7 Alerting

17.7.1 Alert Rules

Prometheus Alertmanager handles alerting:

Listing 17.4: Alert rule example

```yaml
groups:
  - name: api
    rules:
      - alert: HighLatency
        expr: |
          histogram_quantile(0.95,
            rate(request_duration_seconds_bucket[5m])
          ) > 2.5
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "API latency exceeds SLO"
```

17.7.2 Escalation Paths

- Warning: Slack notification
- Critical: PagerDuty alert
- Emergency: Phone escalation

17.8 Debugging Tools

17.8.1 Request Tracing

Every request can be traced end-to-end using the trace ID, showing timing for each component and identifying bottlenecks.

17.8.2 Log Correlation

Logs are correlated by trace ID, enabling quick context gathering when investigating issues.

Chapter 18

Security Architecture

Security is foundational to ARBOR's design, protecting user data, API access, and system integrity. This chapter examines authentication, authorization, data protection, and operational security measures.

18.1 Authentication

18.1.1 Auth0 Integration

Auth0 provides identity management:

- OAuth2/OIDC compliant flows
- Social login (Google, Apple, GitHub)
- Email/password with MFA
- Enterprise SSO (SAML, OIDC)

18.1.2 Token Strategy

Access Tokens Short-lived (15 min), JWT format

Refresh Tokens Long-lived (7 days), secure rotation

API Keys Service-to-service authentication

18.1.3 Token Validation

Listing 18.1: JWT validation

```python
from jose import jwt, JWTError

async def validate_token(token: str) -> TokenClaims:
    try:
        payload = jwt.decode(
            token,
            settings.AUTH0_PUBLIC_KEY,
            algorithms=["RS256"],
            audience=settings.AUTH0_AUDIENCE,
            issuer=f"https://{settings.AUTH0_DOMAIN}/"
        )
        return TokenClaims(**payload)
    except JWTError:
        raise HTTPException(status_code=401)
```

18.2 Authorization

18.2.1 Role-Based Access Control

Roles define permission sets:

Role	Permissions
Anonymous	Public discovery queries
User	Discovery, save entities, feedback
Curator	Validate entities, edit content
Admin	Full system access

Table 18.1: Role definitions

18.2.2 Permission Enforcement

Listing 18.2: Authorization decorator

```python
from functools import wraps

def require_role(required_role: str):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, user: User, **kwargs):
            if required_role not in user.roles:
                raise HTTPException(
                    status_code=403,
                    detail="Insufficient permissions"
                )
            return await func(*args, user=user, **kwargs)
        return wrapper
    return decorator
```

18.3 API Security

18.3.1 Rate Limiting

Tiered rate limits protect against abuse:

Tier	Requests/min	LLM calls/min
Anonymous	10	5
Free User	30	15
Premium	100	50
Enterprise	Custom	Custom

Table 18.2: Rate limit tiers

18.3.2 Input Validation

All inputs are validated:

- Pydantic models for type safety
- Length limits on text fields
- Allowlist validation for enums
- SQL injection prevention

18.4 LLM Security

18.4.1 Prompt Injection Prevention

NeMo Guardrails protects against prompt attacks:

- Input sanitization
- System prompt isolation
- Output validation
- Jailbreak detection

18.4.2 Content Moderation

All LLM outputs pass through moderation:

- Toxicity detection
- PII detection and redaction
- Hallucination checking against knowledge base

18.5 Data Protection

18.5.1 Encryption

At Rest AES-256 for database storage

In Transit TLS 1.3 for all connections

Application Field-level encryption for sensitive data

18.5.2 Data Minimization

- Collect only necessary data
- Automated PII detection
- Retention limits with automatic deletion
- Anonymization for analytics

18.6 Network Security

18.6.1 Network Policies

Kubernetes network policies restrict traffic:

- Default deny ingress
- Explicit allowlists per service
- Database access only from API tier
- Management access via bastion

18.6.2 Web Application Firewall

Cloudflare WAF provides:

- DDoS protection
- OWASP rule set
- Bot management
- Custom rules for API endpoints

18.7 Secrets Management

18.7.1 Secret Storage

Secrets are managed securely:

- HashiCorp Vault for production
- Kubernetes secrets for runtime
- No secrets in code or logs
- Rotation policies

18.7.2 Secret Access

Applications access secrets through:

- Environment variables (injected at deploy)
- Vault sidecar for dynamic secrets
- Kubernetes service accounts

18.8 Audit and Compliance

18.8.1 Audit Logging

Security-relevant events are logged:

- Authentication attempts
- Authorization decisions
- Data access patterns
- Configuration changes

18.8.2 Compliance Considerations

GDPR Data minimization, right to erasure, consent

SOC 2 Access controls, monitoring, encryption

CCPA California privacy requirements

18.9 Security Testing

18.9.1 Ongoing Testing

- Automated vulnerability scanning in CI
- Dependency vulnerability monitoring
- Periodic penetration testing
- Bug bounty program (enterprise)

Part VI

Testing & Quality Assurance

Chapter 19

Test Strategy

ARBOR's test strategy balances comprehensive coverage with development velocity, employing a pyramid of tests from fast unit tests to slower integration and end-to-end validations. This chapter outlines the testing philosophy and infrastructure.

19.1 Testing Philosophy

19.1.1 Guiding Principles

Test Pyramid Many unit tests, fewer integration tests, minimal E2E tests

Fast Feedback Local tests complete in seconds

Determinism Tests produce consistent results

Isolation Tests don't depend on external state

Documentation Tests serve as executable documentation

19.1.2 Coverage Targets

Layer	Target	Current
Unit Tests	80%	78%
Integration Tests	Critical paths	45 scenarios
E2E Tests	Core flows	12 flows

Table 19.1: Test coverage targets

19.2 Test Types

19.2.1 Unit Tests

Unit tests validate individual functions and classes in isolation:

- Fast execution ($<$ 1s per file)
- No external dependencies
- Mocked collaborators
- High coverage target

19.2.2 Integration Tests

Integration tests validate component interactions:

- Real database instances (containerized)
- Service-to-service communication
- Event processing flows
- Medium execution time

19.2.3 End-to-End Tests

E2E tests validate complete user journeys:

- Full system deployment
- API-level interactions
- Browser automation for frontend
- Slowest but highest confidence

19.3 Test Infrastructure

19.3.1 pytest Configuration

pytest serves as the test framework:

Listing 19.1: pytest configuration

```
# pyproject.toml
[tool.pytest.ini_options]
asyncio_mode = "auto"
testpaths = ["tests"]
markers = [
    "unit: Unit tests",
    "integration: Integration tests",
    "e2e: End-to-end tests",
    "slow: Slow-running tests",
]
```

19.3.2 Test Fixtures

Shared fixtures provide consistent test setup:

Listing 19.2: Common fixtures

```
@pytest.fixture
async def db_session():
    """Provide isolated database session."""
    async with async_session_maker() as session:
        yield session
        await session.rollback()

@pytest.fixture
def mock_llm():
    """Mock LLM for deterministic testing."""
    with patch("app.llm.gateway.complete") as mock:
        mock.return_value = MockLLMResponse(
            content="Test response"
        )
        yield mock
```

19.3.3 Test Containers

Testcontainers provides isolated database instances:

Listing 19.3: Testcontainers usage

```
@pytest.fixture(scope="session")
```

```python
def postgres_container():
    with PostgresContainer("postgres:16") as postgres:
        yield postgres

@pytest.fixture(scope="session")
def qdrant_container():
    with DockerContainer("qdrant/qdrant:latest") as qdrant:
        qdrant.with_exposed_ports(6333)
        qdrant.start()
        yield qdrant
```

19.4 CI/CD Integration

19.4.1 Pipeline Stages

1. **Lint**: Code style and type checking

2. **Unit Tests**: Fast feedback on logic

3. **Integration Tests**: Component interactions

4. **E2E Tests**: Full flow validation (staging)

19.4.2 Parallel Execution

Tests execute in parallel for speed:

Listing 19.4: Parallel test execution

```bash
# Unit tests: maximum parallelism
pytest tests/unit -n auto

# Integration tests: limited parallelism
pytest tests/integration -n 4
```

19.5 LLM Testing Considerations

19.5.1 Deterministic Testing

LLM responses are mocked for determinism:

Listing 19.5: LLM mock fixture

```python
@pytest.fixture
def deterministic_llm(monkeypatch):
    """Provide deterministic LLM responses."""
    responses = {
        "intent": '{"intent": "RECOMMENDATION", ...}',
        "synthesis": "Here are my recommendations...",
    }

    async def mock_complete(model, messages, **kwargs):
        prompt_type = detect_prompt_type(messages)
        return MockResponse(content=responses[prompt_type])

    monkeypatch.setattr("app.llm.gateway.complete",
        mock_complete)
```

19.5.2 Golden Set Evaluation

LLM quality is validated against golden sets:

- Curated query-response pairs
- Semantic similarity scoring
- Human evaluation sampling
- Regression detection

19.6 Test Data Management

19.6.1 Factory Pattern

Test data uses factory patterns:

Listing 19.6: Entity factory

```python
class EntityFactory(factory.Factory):
    class Meta:
        model = Entity

    id = factory.LazyFunction(uuid4)
    name = factory.Faker("company")
    domain_id = "lifestyle"
    entity_type = "store"
    vibe_dna = factory.LazyFunction(random_vibe_dna)
```

19.6.2 Seed Data

Consistent seed data enables reproducible integration tests with known entities and relationships.

Chapter 20

Unit Tests

Unit tests form the foundation of ARBOR's test pyramid, providing fast feedback on individual component correctness. This chapter examines unit testing patterns across different system areas.

20.1 Testing Patterns

20.1.1 Arrange-Act-Assert

Tests follow the AAA pattern consistently:

Listing 20.1: AAA pattern example

```python
async def test_entity_creation():
    # Arrange
    factory = EntityFactory()
    data = {"name": "Test Store", "domain_id": "lifestyle"}

    # Act
    entity = factory.create(**data)

    # Assert
    assert entity.name == "Test Store"
    assert entity.domain_id == "lifestyle"
    assert entity.id is not None
```

20.1.2 Given-When-Then

BDD-style for behavior-focused tests:

Listing 20.2: BDD style example

```python
async def test_search_returns_matching_entities():
    """Given entities with matching attributes,
    When searching with those attributes,
    Then matching entities are returned."""

    # Given
    await create_entity(name="Test", category="tailoring")
    await create_entity(name="Other", category="shoes")

    # When
    results = await search(category="tailoring")

    # Then
    assert len(results) == 1
    assert results[0].name == "Test"
```

20.2 Repository Tests

20.2.1 Generic Repository Testing

Listing 20.3: Repository unit tests

```python
class TestGenericEntityRepository:
    async def test_create_entity(self, session):
        repo = GenericEntityRepository(session, config)

        entity = await repo.create({
            "name": "New Entity",
            "domain_id": "lifestyle"
        })

        assert entity.id is not None
        assert entity.name == "New Entity"

    async def test_find_by_filters(self, session):
        repo = GenericEntityRepository(session, config)
        await repo.create({"name": "A", "attributes": {"city":
            "Milan"}})
        await repo.create({"name": "B", "attributes": {"city": "Rome"}})

        results = await repo.get_entities_by_filters(
            filters={"city": "Milan"}
        )

        assert len(results) == 1
        assert results[0]["name"] == "A"
```

20.3 Agent Tests

20.3.1 Intent Router Testing

Listing 20.4: Intent router tests

```
class TestIntentRouter:
    async def test_recommendation_intent(self, mock_llm):
        mock_llm.return_value = MockResponse(
            content='{"intent": "RECOMMENDATION", "confidence": 0.95}'
        )

        router = IntentRouter(llm=mock_llm)
        state = AgentState(user_query="Find cozy restaurants")

        result = await router.analyze(state)

        assert result["intent"].category == "RECOMMENDATION"
        assert result["intent"].confidence > 0.9

    async def test_filter_extraction(self, mock_llm):
        mock_llm.return_value = MockResponse(
            content='{"intent": "RECOMMENDATION", "filters": {"city": "Milan"}}'
        )

        router = IntentRouter(llm=mock_llm)
        state = AgentState(user_query="Tailors in Milan")

        result = await router.analyze(state)

        assert result["extracted_filters"]["city"] == "Milan"
```

20.4 Service Tests

20.4.1 Discovery Service Testing

Listing 20.5: Service layer tests

```
class TestDiscoveryService:
    async def test_search_combines_sources(
        self,
        mock_vector_agent,
        mock_metadata_agent
    ):
        mock_vector_agent.return_value = [entity_a, entity_b]
        mock_metadata_agent.return_value = [entity_b, entity_c]

        service = DiscoveryService(
            vector_agent=mock_vector_agent,
            metadata_agent=mock_metadata_agent
        )

        results = await service.discover("test query")

        # Fusion should combine and deduplicate
        assert len(results) == 3
        assert all(e in results for e in [entity_a, entity_b,
            entity_c])
```

20.5 Utility Tests

20.5.1 Pure Function Testing

Listing 20.6: Utility function tests

```
class TestVibeDNA:
```

```python
    def test_compute_similarity(self):
        vibe_a = VibeDNA(scores={"dim1": 80, "dim2": 60})
        vibe_b = VibeDNA(scores={"dim1": 75, "dim2": 65})

        similarity = compute_vibe_similarity(vibe_a, vibe_b)

        assert 0.9 < similarity < 1.0

    def test_merge_vibes(self):
        vibes = [
            VibeDNA(scores={"dim1": 80}),
            VibeDNA(scores={"dim1": 70}),
        ]

        merged = merge_vibe_dna(vibes)

        assert merged.scores["dim1"] == 75
```

20.6 API Tests

20.6.1 Endpoint Testing

Listing 20.7: API endpoint tests

```python
class TestDiscoveryAPI:
    async def test_discover_endpoint(self, client,
        mock_discovery):
        mock_discovery.return_value = DiscoveryResult(
            message="Test response",
            recommendations=[]
        )

        response = await client.post(
            "/api/v1/discover",
            json={"query": "test"}
        )

        assert response.status_code == 200
        assert response.json()["message"] == "Test response"

    async def test_discover_validation(self, client):
        response = await client.post(
            "/api/v1/discover",
            json={}  # Missing query
        )

        assert response.status_code == 422
```

20.7 Coverage and Quality

20.7.1 Coverage Reporting

pytest-cov generates coverage reports:

Listing 20.8: Coverage commands

```
# Run with coverage
pytest --cov=app --cov-report=html tests/unit

# Enforce minimum coverage
pytest --cov=app --cov-fail-under=80 tests/unit
```

20.7.2 Coverage Exclusions

Certain code is excluded from coverage requirements:

- Abstract base classes
- Type stubs
- Configuration loaders
- CLI entry points

Chapter 21

Integration and End-to-End Tests

Integration tests validate that components work together correctly, while end-to-end tests verify complete user journeys. This chapter examines these higher-level testing approaches.

21.1 Integration Testing

21.1.1 Database Integration

Tests verify correct database interactions:

Listing 21.1: Database integration test

```
@pytest.mark.integration
async def test_entity_lifecycle(db_session, qdrant_client):
    """Test entity creation flows to all databases."""

    # Create entity
    repo = GenericEntityRepository(db_session, config)
    entity = await repo.create({
        "name": "Integration Test",
        "domain_id": "lifestyle"
    })

    # Verify PostgreSQL
    fetched = await repo.get_by_id(entity.id)
    assert fetched is not None

    # Verify Qdrant (after sync)
    await asyncio.sleep(0.5)
    results = await qdrant_client.search(
        collection="entities",
        query_filter={"entity_id": str(entity.id)}
    )
    assert len(results) == 1
```

21.1.2 Agent Integration

Tests verify agent orchestration:

Listing 21.2: Agent integration test

```
@pytest.mark.integration
async def test_discovery_flow(
    db_with_entities,
    qdrant_with_vectors,
    mock_llm
):
    """Test complete discovery flow."""

    # Setup mock LLM responses
    mock_llm.side_effect = [
        intent_response,
        synthesis_response
    ]

    # Execute discovery
    result = await discovery_graph.ainvoke(
        AgentState(user_query="Find tailors in Milan")
    )

    # Verify results
    assert result["intent"].category == "RECOMMENDATION"
    assert len(result["recommendations"]) > 0
    assert "tailor" in result["final_response"].lower()
```

21.2 API Integration Testing

21.2.1 Full Stack API Tests

Listing 21.3: API integration test

```python
@pytest.mark.integration
async def test_discover_with_real_databases(
    test_client,
    seeded_database
):
    """Test discovery endpoint with real databases."""

    response = await test_client.post(
        "/api/v1/discover",
        json={"query": "Find cozy cafes"}
    )

    assert response.status_code == 200
    data = response.json()
    assert "message" in data
    assert "recommendations" in data
    assert len(data["recommendations"]) > 0
```

21.3 End-to-End Testing

21.3.1 User Journey Tests

E2E tests validate complete user flows:

Listing 21.4: E2E user journey

```python
@pytest.mark.e2e
class TestDiscoveryJourney:
    async def test_search_and_detail_flow(self, browser):
        """Test user discovering and viewing entity."""

        # Navigate to discovery
        page = await browser.new_page()
        await page.goto("http://localhost:3000")

        # Enter query
        await page.fill('[data-testid="search-input"]', "tailors")
        await page.press('[data-testid="search-input"]', "Enter")

        # Wait for results
        await page.wait_for_selector('[data-testid="result-card"]')
        results = await page.query_selector_all('[data-testid="result-card"]')
        assert len(results) > 0

        # Click first result
        await results[0].click()

        # Verify detail page
        await page.wait_for_selector('[data-testid="entity-detail"]')
        title = await page.text_content('[data-testid="entity-name"]')
        assert title is not None
```

21.3.2 Playwright Configuration

Listing 21.5: Playwright setup

```
@pytest.fixture(scope="session")
async def browser():
    async with async_playwright() as p:
        browser = await p.chromium.launch(headless=True)
        yield browser
        await browser.close()
```

21.4 Contract Testing

21.4.1 API Contract Validation

OpenAPI schema validates API contracts:

Listing 21.6: Contract test

```python
@pytest.mark.contract
def test_discover_response_schema():
    """Verify response matches OpenAPI schema."""

    response = client.post("/api/v1/discover", json={"query": "test"})

    # Validate against schema
    validate_response(
        response.json(),
        openapi_spec["paths"]["/discover"]["post"]["responses"]["200"]
    )
```

21.5 Performance Baseline Tests

21.5.1 Latency Assertions

Listing 21.7: Performance baseline test

```python
@pytest.mark.performance
async def test_discovery_latency(seeded_database, benchmark):
    """Verify discovery meets latency requirements."""

    async def discover():
        return await discovery_service.discover("test query")

    result = await benchmark.pedantic(
        discover,
        iterations=10,
        rounds=5
    )

    assert benchmark.stats.stats.mean < 2.5  # seconds
```

21.6 Test Environment Management

21.6.1 Docker Compose for Testing

Listing 21.8: Test docker-compose

```
# docker-compose.test.yml
services:
  postgres-test:
    image: postgres:16
    environment:
      POSTGRES_DB: arbor_test

  qdrant-test:
    image: qdrant/qdrant:latest
    ports:
      - "6333:6333"

  neo4j-test:
    image: neo4j:5
    environment:
      NEO4J_AUTH: neo4j/test
```

21.6.2 Environment Isolation

Each test run uses isolated resources to prevent interference between tests.

21.7 Continuous Integration

21.7.1 CI Pipeline Configuration

Listing 21.9: CI pipeline

```
test:
  stage: test
  services:
    - postgres:16
    - qdrant/qdrant:latest
  script:
    - pytest tests/unit -n auto
    - pytest tests/integration --tb=short
    coverage: '/TOTAL.*\s+(\d+%)/'
```

Chapter 22

Chaos and Load Testing

Production reliability requires validation under stress and failure conditions. This chapter examines chaos engineering and load testing approaches for ARBOR.

22.1 Chaos Engineering

22.1.1 Philosophy

Chaos engineering proactively identifies weaknesses by introducing controlled failures:

- Verify failure detection mechanisms work
- Validate graceful degradation behavior
- Test recovery procedures
- Build confidence in system resilience

22.1.2 Chaos Experiments

Experiment	**Validates**
Database failover	Read replica promotion, connection handling
LLM provider outage	Fallback routing, circuit breaker
Network partition	Split-brain handling, timeout behavior
Pod termination	Graceful shutdown, request draining
Resource exhaustion	Memory limits, CPU throttling

Table 22.1: Chaos experiment categories

22.2 Chaos Toolkit

22.2.1 Experiment Definition

Listing 22.1: Chaos experiment definition

```yaml
# experiments/database_failover.yaml
title: Database Failover Resilience
description: Verify system handles database failover

steady-state-hypothesis:
  title: Service remains available
  probes:
    - name: api_responds
      type: probe
      tolerance: 200
      provider:
        type: http
        url: http://api/health

method:
  - name: kill_primary_database
    type: action
    provider:
      type: process
      path: kubectl
      arguments: ["delete", "pod", "postgres-0"]

rollbacks:
  - name: restore_database
    type: action
    provider:
      type: process
      path: kubectl
      arguments: ["rollout", "restart", "statefulset/postgres"]
```

22.3 Load Testing

22.3.1 Load Test Objectives

- Validate performance under expected load
- Identify breaking points (stress testing)
- Verify autoscaling behavior
- Measure degradation characteristics

22.3.2 Load Profiles

Profile	RPS	Duration
Baseline	50	10 min
Peak Load	200	30 min
Stress	500	15 min
Spike	1000 (burst)	2 min
Soak	100	4 hours

Table 22.2: Load test profiles

22.4 Locust Load Testing

22.4.1 Test Scenarios

Listing 22.2: Locust load test

```python
from locust import HttpUser, task, between

class DiscoveryUser(HttpUser):
    wait_time = between(1, 3)

    @task(10)
    def discover(self):
        queries = [
            "Find tailors in Milan",
            "Cozy restaurants nearby",
            "Best coffee shops",
        ]
        self.client.post(
                "/api/v1/discover",
                json={"query": random.choice(queries)}
            )

    @task(3)
    def get_entity(self):
        entity_id = random.choice(self.entity_ids)
        self.client.get(f"/api/v1/entities/{entity_id}")

    @task(1)
    def browse_category(self):
        self.client.get("/api/v1/categories/tailoring")
```

22.5 Performance Metrics

22.5.1 Key Metrics Under Load

Latency P50, P95, P99 response times

Throughput Requests per second sustained

Error Rate Percentage of failed requests

Resource Usage CPU, memory, connections

22.5.2 Acceptance Criteria

Metric	Baseline	Peak	Stress
P95 Latency	< 1.5s	< 2.5s	< 5s
Error Rate	$< 0.1\%$	$< 0.5\%$	$< 2\%$
Throughput	50 rps	200 rps	300 rps

Table 22.3: Performance acceptance criteria

22.6 Database Load Testing

22.6.1 Query Performance

Listing 22.3: Database load test

```python
@pytest.mark.load
async def test_concurrent_queries(db_pool):
    """Test database under concurrent query load."""

    async def run_query():
        async with db_pool.acquire() as conn:
            return await conn.fetch(
                "SELECT * FROM entities WHERE domain_id = $1 LIMIT 50",
                "lifestyle"
            )

    tasks = [run_query() for _ in range(100)]
    results = await asyncio.gather(*tasks)

    assert all(len(r) > 0 for r in results)
```

22.7 LLM Load Testing

22.7.1 Token Budget Testing

Listing 22.4: LLM throughput test

```python
@pytest.mark.load
async def test_llm_throughput(llm_client):
    """Verify LLM can handle concurrent requests."""

    async def make_request():
        return await llm_client.complete(
            model="gpt-4o",
            messages=[{"role": "user", "content": "Test prompt"}],
            max_tokens=100
        )

    start = time.time()
    tasks = [make_request() for _ in range(50)]
    results = await asyncio.gather(*tasks, return_exceptions=True)
    duration = time.time() - start

    errors = [r for r in results if isinstance(r, Exception)]
    assert len(errors) / len(results) < 0.05  # < 5% error rate
```

22.8 Reporting

22.8.1 Load Test Reports

Reports include:

- Latency distribution charts
- Throughput over time
- Error rate by endpoint
- Resource utilization graphs
- Comparison with previous runs

22.8.2 Trend Analysis

Performance trends are tracked across releases to identify regressions early.

Part VII

Appendices

API Reference

This appendix provides a comprehensive reference for ARBOR's REST API endpoints.

.1 API Overview

.1.1 Base URL

Production `https://api.arbor.io/v1`

Staging `https://api.staging.arbor.io/v1`

Development `http://localhost:8000/v1`

.1.2 Authentication

All authenticated endpoints require a Bearer token:

Listing 5: Authentication header

```
Authorization: Bearer <access_token>
```

.2 Discovery Endpoints

.2.1 POST /discover

Execute a discovery query.

Request Body:

```
{
  "query": "Find cozy cafes in Milan",
  "location": {"lat": 45.4642, "lng": 9.1900},
  "session_id": "optional-session-id",
  "filters": {
    "category": "cafe",
    "price_tier": [1, 2, 3]
  }
}
```

Response:

```
{
  "message": "Here are some cozy cafes...",
  "recommendations": [
    {
      "id": "uuid",
      "name": "Cafe Milano",
      "score": 0.92,
      "explanation": "This cafe offers..."
    }
  ],
  "trace_id": "trace-uuid"
}
```

.3 Entity Endpoints

.3.1 GET /entities/{id}

Retrieve entity details.

Response:

```
{
  "id": "uuid",
  "name": "Entity Name",
  "entity_type": "store",
  "attributes": {...},
  "vibe_dna": {"formality": 65, "craftsmanship": 85},
  "location": {"lat": 45.4642, "lng": 9.1900, "address": "..."}
}
```

.3.2 GET /entities

List entities with filtering.

Query Parameters:

- `domain_id`: Filter by domain
- `category`: Filter by category
- `city`: Filter by city
- `limit`: Maximum results (default: 20)
- `offset`: Pagination offset

.4 User Endpoints

.4.1 GET /users/me

Get current user profile. Requires authentication.

.4.2 POST /users/me/saved

Save an entity to user's collection.

Request Body:

```
{
  "entity_id": "uuid"
}
```

.5 Admin Endpoints

.5.1 POST /admin/entities

Create a new entity. Requires admin role.

.5.2 PUT /admin/entities/{id}

Update entity attributes. Requires curator role.

.5.3 POST /admin/entities/{id}/validate

Approve or reject entity validation.

Request Body:

```
{
  "action": "approve",  // or "reject"
  "notes": "optional curator notes"
}
```

.6 Webhook Endpoints

.6.1 POST /webhooks/entity-update

Receive entity update notifications (for integrations).

.6.2 POST /webhooks/feedback

Submit external feedback data.

.7 Error Responses

.7.1 Error Format

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Query is required",
    "details": {...}
  }
}
```

.7.2 HTTP Status Codes

Code	Meaning
200	Success
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
422	Validation Error
429	Rate Limited
500	Internal Error

Table 4: HTTP status codes

.8 Rate Limits

Rate limits are returned in response headers:

Listing 6: Rate limit headers

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 95
X-RateLimit-Reset: 1640000000
```

Configuration Reference

This appendix documents ARBOR's configuration files and environment variables.

.9 Domain Configuration

.9.1 Domain Profile Structure

Domain profiles define domain-specific configuration:

Listing 7: Domain profile example

```yaml
domain:
  id: lifestyle
  name: "Lifestyle & Fashion"
  description: "Curated discovery for lifestyle entities"

categories:
  - id: tailoring
    name: "Tailoring"
    subcategories:
      - id: bespoke
        name: "Bespoke"
      - id: mtm
        name: "Made-to-Measure"
  - id: shoes
    name: "Footwear"

dimensions:
  formality:
    name: "Formality"
    description: "0 = Casual, 100 = Formal"
    weight: 1.0
  craftsmanship:
    name: "Craftsmanship"
    description: "0 = Mass-produced, 100 = Artisanal"
    weight: 1.2

styles:
  - id: neapolitan
    name: "Neapolitan"
    keywords: ["spalla scesa", "unconstructed"]
  - id: english
    name: "English/Savile Row"
    keywords: ["structured", "padded shoulder"]

relationship_types:
  - sells_brand
  - trained_by
  - has_style

curator_persona:
  name: "The Style Curator"
  voice: "Sophisticated but approachable"
```

.10 Source Schema Configuration

.10.1 Data Source Schema

SOURCE_SCHEMA_CONFIG defines data source mappings:

Listing 8: Source schema configuration

```
source_schema:
  name: "google_maps"
  entity_mapping:
    name: "$.name"
    external_id: "$.place_id"
    location:
      lat: "$.geometry.location.lat"
      lng: "$.geometry.location.lng"
    attributes:
      rating: "$.rating"
      review_count: "$.user_ratings_total"
      price_level: "$.price_level"

  category_mapping:
    "clothing_store": "fashion"
    "restaurant": "dining"
    "cafe": "coffee"
```

.11 Environment Variables

.11.1 Core Settings

Variable	Description
`ENVIRONMENT`	Environment name (dev/staging/prod)
`DEBUG`	Enable debug mode
`LOG_LEVEL`	Logging level
`SECRET_KEY`	Application secret key

Table 5: Core environment variables

.11.2 Database Settings

Variable	Description
`DATABASE_URL`	PostgreSQL connection string
`QDRANT_URL`	Qdrant server URL
`QDRANT_API_KEY`	Qdrant API key
`NEO4J_URI`	Neo4j connection URI
`NEO4J_USER`	Neo4j username
`NEO4J_PASSWORD`	Neo4j password
`REDIS_URL`	Redis connection string

Table 6: Database environment variables

.11.3 LLM Settings

Variable	Description
`OPENAI_API_KEY`	OpenAI API key
`ANTHROPIC_API_KEY`	Anthropic API key
`AZURE_API_KEY`	Azure OpenAI key
`AZURE_API_BASE`	Azure endpoint URL
`DEFAULT_LLM_MODEL`	Default model for inference

Table 7: LLM environment variables

.11.4 Authentication

Variable	Description
`AUTH0_DOMAIN`	Auth0 tenant domain
`AUTH0_AUDIENCE`	Auth0 API audience
`AUTH0_CLIENT_ID`	Auth0 client ID
`AUTH0_CLIENT_SECRET`	Auth0 client secret

Table 8: Authentication environment variables

.12 Prompt Templates

.12.1 Template Directory

Prompts are stored in `config/prompts/`:

Listing 9: Prompt directory structure

```
config/prompts/
|-- intent_classification.txt
|-- curator_synthesis.txt
|-- vibe_extraction.txt
|-- vision_analysis.txt
`-- entity_comparison.txt
```

.12.2 Template Variables

Common template variables:

- `{domain_name}`: Current domain name
- `{dimensions}`: List of vibe dimensions
- `{categories}`: Available categories
- `{query}`: User query
- `{context}`: Retrieved context
- `{persona}`: Curator persona

.13 Feature Flags

Feature flags control runtime behavior:

Listing 10: Feature flags

```
features:
  semantic_cache: true
  graph_rag: true
  personalization: true
  experimental_reranker: false
  new_curator_prompt: false
```

Glossary

This glossary defines key terms used throughout the ARBOR documentation.

.14 Architecture Terms

Agentic Orchestration The pattern of using autonomous AI agents coordinated by a supervisor to accomplish complex tasks, as implemented via LangGraph in ARBOR.

GraphRAG Graph-based Retrieval-Augmented Generation; using knowledge graph traversal to provide context for LLM responses.

Hybrid Search Combining dense vector similarity (semantic) and sparse vector matching (keyword) for retrieval, using RRF fusion.

Knowledge Trinity ARBOR's three-database architecture: PostgreSQL for structured data, Qdrant for vector search, Neo4j for graph relationships.

LLM Gateway Centralized layer managing LLM access, including routing, caching, guardrails, and observability.

Polyglot Persistence Using multiple database technologies, each optimized for specific access patterns.

.15 Domain Concepts

Curator Human expert who validates entities and maintains knowledge base quality; also the AI persona that synthesizes recommendations.

Domain A configured vertical such as lifestyle, hospitality, or real estate, with its own categories, dimensions, and relationship types.

Entity Any discoverable item in the knowledge base: a store, restaurant, professional, brand, or other domain-specific object.

Vibe DNA Multi-dimensional scoring representing an entity's qualitative characteristics (e.g., formality, craftsmanship, price level).

Vibe Dimension A single axis of the Vibe DNA, defined per domain (e.g., "formality: 0-100").

.16 Technical Components

Agent State The TypedDict structure containing all information that flows through the agent graph during query processing.

Curator Agent The agent responsible for synthesizing retrieval results into natural language recommendations.

Enrichment Orchestrator Component that coordinates AI analysis (vision, review, embedding) during entity ingestion.

GenericEntityRepository Schema-agnostic repository pattern enabling domain flexibility without code changes.

Historian Agent Agent that queries Neo4j for relationship context around retrieved entities.

Intent Router Agent that classifies user queries and extracts structured filters.

LangGraph LangChain framework for building stateful, multi-agent workflows as graphs.

Langfuse LLM observability platform for tracing, cost tracking, and evaluation.

LiteLLM Multi-provider LLM gateway enabling unified access to OpenAI, Anthropic, Azure, etc.

Metadata Agent Agent that queries PostgreSQL for entities matching structured filters.

NeMo Guardrails NVIDIA framework for adding safety rails to LLM applications.

Qdrant Vector database used for semantic similarity search.

Reciprocal Rank Fusion (RRF) Algorithm for combining ranked lists from multiple retrieval sources.

Temporal.io Durable workflow execution platform used for ingestion and background processing.

Vector Agent Agent that performs semantic similarity search in Qdrant.

.17 ML Terms

A/B Testing Controlled experimentation comparing variants to measure impact.

Causal Inference Statistical methods for estimating treatment effects.

Drift Detection Monitoring for changes in data distributions that may affect model performance.

Feature Store Real-time serving infrastructure for ML features.

Knowledge Distillation Compressing large models into smaller, efficient variants.

Reranking Second-stage ranking refinement after initial retrieval.

RLHF Reinforcement Learning from Human Feedback; using human preferences to improve model outputs.

SHAP SHapley Additive exPlanations; method for explaining individual predictions.

.18 Infrastructure Terms

Circuit Breaker Pattern that prevents cascading failures by stopping requests to failing services.

HPA Horizontal Pod Autoscaler; Kubernetes mechanism for scaling based on metrics.

Istio Service mesh providing mTLS, traffic management, and observability.

OpenTelemetry Observability framework for distributed tracing and metrics.

PgBouncer Connection pooler for PostgreSQL.

StatefulSet Kubernetes workload for stateful applications like databases.

.19 Abbreviations

Abbreviation	Meaning
AAA	Arrange-Act-Assert
API	Application Programming Interface
CDC	Change Data Capture
CDN	Content Delivery Network
CI/CD	Continuous Integration/Continuous Deployment
E2E	End-to-End
GNN	Graph Neural Network
JWT	JSON Web Token
LLM	Large Language Model
OIDC	OpenID Connect
P95	95th Percentile
RAG	Retrieval-Augmented Generation
RBAC	Role-Based Access Control
RPS	Requests Per Second
SLI/SLO	Service Level Indicator/Objective
SSR	Server-Side Rendering
WAF	Web Application Firewall

Table 9: Common abbreviations