USER GUIDE

Locality Engine v2.0

PRACTICAL EXAMPLES & API REFERENCE

Quick Start Guide

Get up and running with Locality Engine in under 5 minutes.

Basic Usage - Your First Transformation

Import Locality Engine and transform a message to be culturally appropriate.

```
# Import the engine and cultural frameworks from locality_engine import LocalityEngine, CulturalFramework # Initialize engine for Bantu/African context engine = LocalityEngine(culture=CulturalFramework.BANTU_COLLECTIVE) # Transform a direct message to be more dignified original = "What do you want?" transformed = engine.transform(original) print(f"Original: {original}") print(f"Transformed: {transformed}")
```

```
Output:
Original: What do you want?
Transformed: What would you like me to assist you with?
```

Core API Reference

`LocalityEngine(culture, active_factors=None)`

Initialize the Locality Engine with a specific cultural framework.

Parameters:

`culture` (CulturalFramework): Cultural framework to apply (BANTU_COLLECTIVE, AMERICAN_INDIVIDUALIST, INDONESIAN_PANCASILA, JAPANESE_WA)

`active_factors` (List[int], optional): List of factor numbers (1-18) to activate. If None, all factors are active.

Returns:

LocalityEngine instance configured for the specified culture

`transform(message, mode=TransformMode.CONTEXT_AWARE)`

Main transformation method. Adapts message based on cultural context.

Parameters:

`message` (str): Input text to transform

`mode` (TransformMode): Transformation strategy - DETERMINISTIC (rule-based), PROBABILISTIC (natural variation), or CONTEXT_AWARE (intelligent auto-selection, default)

Returns:

str: Culturally-adapted message preserving dignity and meaning

`assess_dignity_risk(message)`

Analyzes message for potential dignity violations across all 18 Cultural Reasoning factors.

Parameters:

`message` (str): Text to assess for cultural risks

Returns:

dict: Contains risk_score (0-100), risk_level (low/medium/high/critical), violated_factors (list), recommendation, and culture

`get_active_factors()`

Returns list of currently active Cultural Reasoning factors with their properties.

Returns:

List[dict]: Active factors with number, name, weight, description, and active status

Practical Examples

Example 1: AI Chatbot Integration

Adapt customer service chatbot responses based on user culture.

```
from locality_engine import LocalityEngine, CulturalFramework # Detect user culture (simplified - in production, use proper detection) user_culture = CulturalFramework.INDONESIAN_PANCASILA # Initialize engine for user's culture engine = LocalityEngine(culture=user_culture) # Chatbot's original response (too direct) bot_response = "I disagree with your assessment. That approach won't work." # Transform to be culturally appropriate adapted_response = engine.transform(bot_response) print(f"Original: {bot_response}") print(f"Adapted: {adapted_response}")
```

```
Output:
Original: I disagree with your assessment. That approach won't work.
Adapted: Perhaps we could find harmony in both views. That approach might face challenges.
```

Example 2: Healthcare Communication

Adjust medical questions to account for cultural pain communication patterns.

```
from locality_engine import LocalityEngine, CulturalFramework # Initialize for Bantu/African patient engine = LocalityEngine(culture=CulturalFramework.BANTU_COLLECTIVE) # Standard Western medical question doctor_question = "Tell me your pain level. Are you in pain?" # Adapt to account for Factor #2 (Honor - pain understatement) adapted_question = engine.transform(doctor_question) # Also assess if question has dignity risks risk_assessment = engine.assess_dignity_risk(doctor_question) print(f"Original: {doctor_question}") print(f"Adapted: {adapted_question}") print(f"Risk: {risk_assessment['risk_score']}/100") print(f"Recommendation: {risk_assessment['recommendation']}")
```

💡 Healthcare Context

In many cultures, patients understate pain to maintain dignity (Factor #2: Honor). The adapted question should be paired with proactive pain management rather than waiting for patient requests. This approach prevents the 2-3X maternal mortality gap.

Example 3: Business Proposal Screening

Scan business proposals for cultural violations before sending to international partners.

```
from locality_engine import LocalityEngine, CulturalFramework # Prepare engines for different target audiences japanese_engine =
```

```
LocalityEngine(culture=CulturalFramework.JAPANESE_WA)
indonesian_engine =
LocalityEngine(culture=CulturalFramework.INDONESIAN_PANCASILA) #
Original proposal (too direct for Asian markets) proposal = """ We
need an immediate decision on this partnership. Your current approach
is inefficient and needs to change. Our solution is clearly superior.
""" # Assess dignity risks for each culture japanese_risk =
japanese_engine.assess_dignity_risk(proposal) indonesian_risk =
indonesian_engine.assess_dignity_risk(proposal) print("JAPANESE
MARKET") print(f"Risk Score: {japanese_risk['risk_score']}/100
({japanese_risk['risk_level']})") print(f"Violated Factors:
{len(japanese_risk['violated_factors'])}") print() print("INDONESIAN
MARKET") print(f"Risk Score: {indonesian_risk['risk_score']}/100
({indonesian_risk['risk_level']})") print(f"Recommendation:
{indonesian_risk['recommendation']}")
```

```
Output:
JAPANESE MARKET
Risk Score: 65.0/100 (high)
Violated Factors: 3

INDONESIAN MARKET
Risk Score: 58.0/100 (high)
Recommendation: TRANSFORM REQUIRED
```

Example 4: Multi-Culture Support (Advanced)

Transform the same message for multiple cultural contexts simultaneously.

```
from locality_engine import LocalityEngine, CulturalFramework,
TransformMode # Original message message = "Your performance needs
improvement. You're not meeting expectations." # Create engines for
each culture cultures = { "Bantu/African":
CulturalFramework.BANTU_COLLECTIVE, "American":
```

```
CulturalFramework.AMERICAN_INDIVIDUALIST, "Indonesian":
CulturalFramework.INDONESIAN_PANCASILA, "Japanese":
CulturalFramework.JAPANESE_WA } print(f"ORIGINAL
MESSAGE:\n{message}\n") print("=" * 70) # Transform for each culture
for culture_name, culture_framework in cultures.items(): engine =
LocalityEngine(culture=culture_framework) transformed =
engine.transform(message, mode=TransformMode.DETERMINISTIC)
print(f"\n{culture_name.upper()} ADAPTATION:") print(transformed)
```

Example 5: Selective Factor Activation

Activate only specific Cultural Reasoning factors for targeted transformations.

```
from locality_engine import LocalityEngine, CulturalFramework # Focus
only on dignity (Factor 1) and honor (Factor 2) engine =
LocalityEngine( culture=CulturalFramework.BANTU_COLLECTIVE,
active_factors=[1, 2] # Only Meliorism and Honor ) # Check which
factors are active active = engine.get_active_factors()
print(f"Active factors: {len(active)}") for factor in active:
print(f" - Factor #{factor['number']}: {factor['name']} (weight:
{factor['weight']})") # Transform message message = "You're wrong
about this. That's a bad idea." transformed =
engine.transform(message) print(f"\nOriginal: {message}")
print(f"Transformed: {transformed}")
```

Transformation Modes Explained

Mode	Description	Best For
DETERMINISTIC	Rule-based transformations with predictable, consistent results. Same input always produces same output.	Official communications, legal documents, medical records, contracts where consistency is critical
PROBABILISTIC	Natural variation using culturally-appropriate alternatives. Introduces randomness to prevent robotic patterns.	Chatbots, customer service, conversational AI, marketing content where variety improves engagement
CONTEXT_AWARE	Intelligent automatic selection. Analyzes message content and chooses optimal transformation strategy.	General-purpose applications, mixed content types, when you want optimal results without manual mode selection

🎯 Recommendation

Use **CONTEXT_AWARE** mode (default) for most applications. It intelligently selects the best strategy based on message content. Use **DETERMINISTIC** when consistency is critical, and **PROBABILISTIC** for natural conversational variety.

Best Practices

✅ DO: Initialize Engine Once

Create a single LocalityEngine instance per culture and reuse it throughout your application. Initialization loads cultural configurations, so creating new instances repeatedly wastes resources.

```
# Good: Create once, use many times engine = LocalityEngine(culture=CulturalFramework.BANTU_COLLECTIVE) for message in messages: transformed = engine.transform(message)
```

✅ DO: Assess Risk Before Important Communications

Always use assess_dignity_risk() before sending high-stakes communications (job offers, medical diagnoses, business proposals). A risk score above 40 indicates transformation is recommended.

```
risk = engine.assess_dignity_risk(message) if risk['risk_score'] > 40: message = engine.transform(message) send_message(message)
```

✅ DO: Combine with User Preferences

Store user cultural preferences and apply appropriate engine. Allow users to opt-in or opt-out of cultural adaptation. Respect user autonomy while offering dignity-preserving defaults.

❌ DON'T: Assume Culture from Demographics

Never assume cultural framework based solely on race, nationality, or ethnicity. Allow users to self-identify or use behavioral signals. Cultural frameworks are about values and decision-making patterns, not stereotypes.

❌ DON'T: Over-Transform Casual Communication

Not every message needs transformation. Casual, low-stakes communication between peers can remain direct. Reserve transformation for high-stakes contexts or when dignity is at risk.

✅ DO: Log Transformations for Audit

In regulated industries (healthcare, finance, legal), log original and transformed messages with transformation rationale. This provides audit trail and demonstrates cultural intelligence compliance.

```
import json risk = engine.assess_dignity_risk(original_message)
transformed = engine.transform(original_message) audit_log = {
"timestamp": datetime.now().isoformat(), "original": original_message,
"transformed": transformed, "culture": engine.culture.value,
"risk_score": risk['risk_score'], "violated_factors":
risk['violated_factors'] } log_to_file(json.dumps(audit_log))
```

Integration Patterns

Pattern 1: Pre-Processing Layer

Add Locality Engine as a pre-processing step before sending messages to users.

```
# Example: Email notification system def send_email(recipient,
subject, body): # Detect recipient culture (from profile/preferences)
user_culture = get_user_culture(recipient) # Initialize engine for
user's culture engine = LocalityEngine(culture=user_culture) #
Transform subject and body adapted_subject =
engine.transform(subject) adapted_body = engine.transform(body) #
Send adapted message email_service.send( to=recipient,
subject=adapted_subject, body=adapted_body )
```

Pattern 2: Real-Time Chatbot Adaptation

Adapt chatbot responses in real-time based on conversation context.

```
class CulturalChatbot: def __init__(self,
default_culture=CulturalFramework.AMERICAN_INDIVIDUALIST):
self.engines = { culture: LocalityEngine(culture=culture) for culture
in CulturalFramework } self.default_culture = default_culture def
respond(self, user_message, user_culture=None): # Select appropriate
engine culture = user_culture or self.default_culture engine =
self.engines[culture] # Generate response (your AI logic here)
raw_response = self.generate_response(user_message) # Adapt response
culturally adapted_response = engine.transform(raw_response) return
adapted_response def generate_response(self, message): # Your chatbot
AI logic here return "Here is my response to your query."
```

Pattern 3: Content Moderation Pipeline

Screen content before publication to prevent cultural violations.

```
def moderate_content(content, target_cultures): """ Screen content for cultural violations before publication. Returns approval status and suggestions. """ results = {} max_risk = 0 for culture_name, culture_framework in target_cultures.items(): engine = LocalityEngine(culture=culture_framework) risk = engine.assess_dignity_risk(content) results[culture_name] = { "risk_score": risk['risk_score'], "risk_level": risk['risk_level'], "violations": risk['violated_factors'], "suggested_adaptation": engine.transform(content) if risk['risk_score'] > 40 else None } max_risk = max(max_risk, risk['risk_score']) return { "approved": max_risk < 40, "max_risk": max_risk, "culture_results": results } # Usage content = "Your approach is completely wrong and inefficient." target_cultures = { "Japanese": CulturalFramework.JAPANESE_WA, "Indonesian": CulturalFramework.INDONESIAN_PANCASILA } moderation_result = moderate_content(content, target_cultures) print(f"Approved: {moderation_result['approved']}") print(f"Max Risk: {moderation_result['max_risk']}/100")
```

Pattern 4: API Wrapper

Create a REST API endpoint for cultural transformation as a service.

```
from fastapi import FastAPI, HTTPException from pydantic import BaseModel from locality_engine import LocalityEngine, CulturalFramework, TransformMode app = FastAPI(title="Locality Engine API") # Initialize engines for all cultures engines = { culture.value: LocalityEngine(culture=culture) for culture in CulturalFramework } class TransformRequest(BaseModel): message: str
```

```
culture: str mode: str = "auto" class TransformResponse(BaseModel): original: str transformed: str culture: str risk_score: float @app.post("/transform", response_model=TransformResponse) def transform_message(request: TransformRequest): if request.culture not in engines: raise HTTPException(status_code=400, detail="Invalid culture") engine = engines[request.culture] # Assess risk risk = engine.assess_dignity_risk(request.message) # Transform mode_map = { "deterministic": TransformMode.DETERMINISTIC, "probabilistic": TransformMode.PROBABILISTIC, "auto": TransformMode.CONTEXT_AWARE } transformed = engine.transform( request.message, mode=mode_map.get(request.mode, TransformMode.CONTEXT_AWARE) ) return TransformResponse( original=request.message, transformed=transformed, culture=request.culture, risk_score=risk['risk_score'] ) # Run with: uvicorn api:app --reload
```

Cultural Framework Reference

Framework	Key Characteristics	Emphasized Factors
BANTU_COLLECTIVE	Ubuntu/Meliorism foundation, collectivist decision-making, extended family obligations, elder consultation, indirect communication	Factor #1 (Meliorism: 0.25), Factor #5 (Family: 0.12), Factor #7 (Post-Colonial Trust: 0.10)

Framework	Key Characteristics	Emphasized Factors
AMERICAN_INDIVIDUALIST	Individual autonomy, direct communication, honor-consciousness for minorities, code-switching awareness	Factor #2 (Honor: 0.20), Factor #7 (Trust for African Americans: 0.08), Factor #4 (Linguistic Tax: 0.10)
INDONESIAN_PANCASILA	Harmony emphasis (Pancasila), Gotong royong collaboration, Bapak-ism authority, malu (shame) sensitivity, musyawarah consensus	Factor #1 (Harmony: 0.22), Factor #11 (Authority: 0.08), Factor #13 (Relational: 0.06)
JAPANESE_WA	Wa (harmony) preservation, Honne/Tatemae communication, Sempai-Kohai hierarchy, indirect correction, collective shame avoidance	Factor #1 (Wa: 0.23), Factor #16 (Collective Shame: 0.05), Factor #17 (Indirect: 0.04)

🌍 Universal Framework

All 18 Cultural Reasoning factors operate across all cultures—they're just weighted differently. Factor #1 (Meliorism/Dignity) is universal: every culture values dignity preservation, but expresses it differently (Ubuntu in Bantu, Wa in Japanese, Harmony in Indonesian, Respect in American).

Performance & Optimization

⚡ Engine Initialization

Engine initialization is lightweight (~10ms). However, for high-throughput applications, initialize once and cache:

```
# Singleton pattern for production class EngineCache: _instances = {} @classmethod def get_engine(cls, culture: CulturalFramework): if culture not in cls._instances: cls._instances[culture] = LocalityEngine(culture=culture) return cls._instances[culture] # Usage engine = EngineCache.get_engine(CulturalFramework.BANTU_COLLECTIVE)
```

⚡ Transformation Performance

Transformation speed depends on mode:

DETERMINISTIC: ~0.5-2ms per message (string replacement)
PROBABILISTIC: ~0.1-0.5ms per message (random selection)
CONTEXT_AWARE: ~1-3ms per message (analysis + transformation)

All modes are fast enough for real-time applications (chat, APIs, live translation).

⚡ Batch Processing

For large-scale content transformation (migrating documentation, bulk email), process in batches:

```
def batch_transform(messages, culture, batch_size=1000): engine = LocalityEngine(culture=culture) results = [] for i in range(0,
```

```
len(messages), batch_size): batch = messages[i:i+batch_size]
batch_results = [engine.transform(msg) for msg in batch]
results.extend(batch_results) # Optional: Progress reporting
print(f"Processed {min(i+batch_size, len(messages))}/{len(messages)}")
return results
```

Troubleshooting

Issue: Transformations Too Aggressive

Problem: Every message is being transformed, even casual low-stakes communication.

Solution: Use risk assessment threshold. Only transform when risk score is meaningful:

```
def smart_transform(message, engine, threshold=30): risk =
engine.assess_dignity_risk(message) if risk['risk_score'] >
threshold: return engine.transform(message) else: return message #
Keep original for low-risk messages
```

Issue: Wrong Culture Applied

Problem: User receives inappropriate cultural adaptation.

Solution: Implement user preference system with explicit opt-in:

```
# User profile with cultural preference user_profile = { "user_id":
"123", "cultural_preference": "bantu_collective",
"enable_adaptation": True, # Explicit opt-in "adaptation_threshold":
40 # Only transform high-risk } def get_user_engine(user_profile): if
not user_profile.get('enable_adaptation', False): return None # User
opted out culture_map = { "bantu_collective":
CulturalFramework.BANTU_COLLECTIVE, "american":
CulturalFramework.AMERICAN_INDIVIDUALIST, "indonesian":
CulturalFramework.INDONESIAN_PANCASILA, "japanese":
CulturalFramework.JAPANESE_WA } culture =
culture_map.get(user_profile['cultural_preference']) return
LocalityEngine(culture=culture) if culture else None
```

Issue: Losing Message Meaning

Problem: Transformed messages lose critical information or become unclear.

Solution: Use DETERMINISTIC mode for critical communications, verify output:

```
# For critical messages, use deterministic and verify def
safe_transform(message, engine): original_length =
len(message.split()) transformed = engine.transform(message,
mode=TransformMode.DETERMINISTIC) transformed_length =
len(transformed.split()) # Sanity check: if message exploded in
length, something's wrong if transformed_length > original_length *
```

```
2: return message # Keep original if transformation seems wrong
return transformed
```

Advanced Usage

Custom Factor Weights

Advanced users can create custom cultural profiles by modifying factor weights:

```
from locality_engine import LocalityEngine, CulturalFramework,
CULTURAL_FACTORS # Create engine engine =
LocalityEngine(culture=CulturalFramework.BANTU_COLLECTIVE) #
Customize factor weights for specific context # Example: Healthcare
context - emphasize Factor #2 (Honor/Pain) engine.factors[2].weight =
0.25 # Increase honor sensitivity engine.factors[4].weight = 0.15 #
Increase linguistic accessibility # Now transformations will
prioritize these factors medical_question = "Tell me your pain
level." adapted = engine.transform(medical_question)
```

⚠️ Warning

Custom factor weights should only be adjusted by users who understand the Cultural Reasoning Framework deeply. Incorrect weights can produce

inappropriate transformations. The default weights are research-validated and appropriate for most use cases.

Multi-Language Support (Future)

While Locality Engine v2.0 operates in English, the framework supports multi-language adaptation:

```
# Future API (v3.0+) engine = LocalityEngine( culture=CulturalFramework.INDONESIAN_PANCASILA, language="id" # Indonesian language ) # Will transform in Bahasa Indonesia message = "Saya tidak setuju dengan pendekatan Anda." transformed = engine.transform(message) # Output: "Mungkin kita bisa menemukan harmoni dalam kedua pandangan."
```

🚀 Coming Soon

Multi-language support is planned for Locality Engine v3.0. The current version (v2.0) operates in English but can be extended by users who provide language-specific transformation rules.

Getting Help

📧 Contact Support

For technical support, feature requests, or consulting on cultural intelligence integration:

Email: viantekeliana@gmail.com
GitHub: @viantekeliana

Response time: 24-48 hours for technical inquiries

📚 Additional Resources

Owner's Manual: Complete product overview with 20+ use cases
Installation Guide: Step-by-step setup instructions
GitHub Repository: Source code, examples, and issue tracking
Research Papers: Vianté Cultural Reasoning Framework™ documentation
API Documentation: Complete API reference (this guide)

All documentation available at github.com/viantekeliana/locality-engine

🤝 Contributing

Locality Engine is currently proprietary software, but contributions are welcome:

Bug Reports: Submit via GitHub Issues
Feature Requests: Email with detailed use case
Cultural Consultations: Help us expand to new cultures
Translation Support: Multi-language transformation rules

Contributors will be credited in documentation and release notes.

Frequently Asked Questions

Q: Does Locality Engine work with non-English text?

A: Version 2.0 is optimized for English. The Cultural Reasoning Framework is language-agnostic, but transformation rules are currently English-only. Multi-language support is planned for v3.0. Users can extend the engine with custom language rules.

Q: How accurate are the cultural transformations?

A: Transformations are based on 10+ years of ethnographic research across 1,100+ documented interactions. Default configurations are research-validated. However, cultural communication is complex—review high-stakes transformations before deployment.

Q: Can I use this in commercial products?

A: Yes, with proper licensing. Contact viantekeliana@gmail.com for commercial licensing terms. Pricing varies based on deployment scale (API calls, user count, industry).

Q: What's the difference between Locality Engine and translation?

A: Translation converts language. Locality Engine adapts *cultural context* while preserving language. It changes communication style, formality, directness, and framing based on cultural values—not just words. Think "cultural compiler" not "translator."

Q: How do I detect user culture automatically?

A: Locality Engine doesn't include culture detection (by design—assumes stereotyping). Recommended approaches: (1) Ask users explicitly, (2) Use behavioral signals (communication patterns over time), (3) Allow users to select from profiles. Never assume culture from demographics alone.

Q: What if user culture isn't one of the four frameworks?

A: The four frameworks are archetypal patterns. Most cultures map to these with weight adjustments. For example, Korean culture → JAPANESE_WA with custom weights. Nigerian Yoruba → BANTU_COLLECTIVE with modifications. Contact us for help mapping your target culture to the framework.

LOCALITY ENGINE v2.0 - USER GUIDE

Created by **Rodney Manyakaidze** (Keliana Vianté)

viantekeliana@gmail.com

GitHub: @viantekeliana

Built on the Vianté Cultural Reasoning Framework™

10+ Years Research · 1,100+ Interactions · 18 Universal Factors

"Deal with matters without anyone losing face or being stripped of dignity."

— The Ubuntu Principle