Parametric Insurance MGA Implementation Guide with VeriCore

> **Building "Atlas Parametric" - An EO-Driven Parametric Insurance MGA**

This comprehensive guide shows you how to build your parametric insurance MGA solution using VeriCore as the trust and integrity foundation. This implementation covers SAR flood, heatwave, solar attenuation, hurricane, and drought parametric products with instant payouts and objective EO triggers.

Executive Summary

What You're Building:

- A parametric insurance MGA (Managing General Agent) platform
- EO-driven triggers (SAR, NDVI, AOD, LST, InSAR)
- Automated 24-72 hour payouts
- Multi-provider EO data acceptance
- Tamper-proof trigger verification
- Regulatory-compliant audit trails

Why VeriCore:

- **Trust Foundation**: Verifiable Credentials for EO data integrity
- **Multi-Provider Support**: Accept data from ESA, Planet, NASA, NOAA without custom integrations
- **Blockchain Anchoring**: Tamper-proof trigger records for regulatory compliance
- **Standardization**: W3C-compliant format works across all providers
- **Automation**: Enable instant payouts with verifiable triggers

Architecture Overview

```
┌─────────────────────────────────────────────────────────────────┐
│                   Atlas Parametric MGA Platform                 │
├─────────────────────────────────────────────────────────────────┤
│                                                                   │
│  ┌──────────────────┐  ┌──────────────────┐  ┌─────────────────┐ │
│  │  EO Data         │  │  Pricing         │  │  Payout         │ │
│  │  Ingestion       │  │  Engine          │  │  Automation     │ │
│  │  (SAR, NDVI,     │  │  (Actuarial)     │  │  (Banking API)  │ │
│  │   AOD, LST)      │  │                  │  │                 │ │
│  └────────┬─────────┘  └────────┬─────────┘  └────────┬────────┘ │
│           │                     │                      │          │
│           └─────────────────────┼──────────────────────┘          │
│                                 │                                   │
│                       ┌─────────▼─────────┐                        │
│                    │  VeriCore Trust Layer │                   │
│                    │  ┌─────────────────┐  │                   │
│                    │  │ DID Management  │  │                   │
│                    │  │ VC Issuance     │  │                   │
│                    │  │ VC Verification │  │                   │
│                    │  │ Blockchain Anchor│  │                   │
│                    │  │ Data Integrity  │  │                   │
│                    │  └─────────────────┘  │                   │
│                    └───────────┬───────────┘                   │
│                                │                               │
│  ┌─────────────────┐  ┌────────▼────────┐  ┌─────────────────┐ │
│  │  Broker Portal  │  │  Trigger        │  │  Reinsurer      │ │
│  │  (Distribution) │  │   Engine        │  │   Dashboard     │ │
│  └─────────────────┘  └─────────────────┘  └─────────────────┘ │
│                                                                │
└────────────────────────────────────────────────────────────────┘
```

Core Components

1. VeriCore Trust Layer

VeriCore provides:

- **DID Management**: Identity for insurers, EO providers, reinsurers, brokers
- **Verifiable Credentials**: EO data wrapped in VCs for integrity
- **Blockchain Anchoring**: Tamper-proof trigger records
- **Multi-Provider Support**: Accept EO data from any certified provider

2. EO Data Ingestion Pipeline

Processes:

- Sentinel-1 SAR (flood detection)
- MODIS/VIIRS LST (heatwave)
- AOD + irradiance (solar attenuation)
- NDVI (drought/agriculture)
- GPM/IMERG (rainfall)
- InSAR (deformation)

3. Trigger Engine

Evaluates parametric triggers:

- Real-time EO data ingestion
- Threshold evaluation
- Tiered payout calculation
- Automatic trigger verification

4. Pricing Engine

Actuarial pricing:

- EO climate archive (20-40 years)
- Hazard-frequency modeling
- Geographic risk scoring
- Reinsurer-approved rates

5. Payout Automation

Automated payouts:

- KYC/AML integration
- Banking API integration
- Reinsurer notification
- Audit trail generation

Implementation: Product Suite

Product 1: SAR-Based Flood Parametric

Data Sources: Sentinel-1 SAR + DEM **Markets:** US (NC, SC, FL, GA) **Payout Range:** $25k - $5M **Triggers:** Depth thresholds (20cm, 50cm, 1m)

Implementation

```kotlin
package com.atlasparametric.products.flood

import com.geoknoesis.vericore.VeriCore
import com.geoknoesis.vericore.contract.models.*
import com.geoknoesis.vericore.core.*
import com.geoknoesis.vericore.json.DigestUtils
import kotlinx.serialization.json.buildJsonObject
import kotlinx.serialization.json.put
import java.time.Instant

/**
 * SAR Flood Parametric Product using Smart Contracts
 *
 * Uses Sentinel-1 SAR data to detect flood depth and trigger automatic payouts
 */
class SarFloodProduct(
    private val vericore: VeriCore,
    private val eoProviderDid: String
) {
    /**
     * Create a flood insurance contract
     */
    suspend fun createFloodContract(
        insurerDid: String,
        insuredDid: String,
        coverageAmount: Double,
        location: Location
    ): SmartContract {

        val contract = vericore.contracts.draft(
            request = ContractDraftRequest(
                contractType = ContractType.Insurance,
                executionModel = ExecutionModel.Parametric(
                    triggerType = TriggerType.EarthObservation,
                    evaluationEngine = "parametric-insurance"
                ),
                parties = ContractParties(
                    primaryPartyDid = insurerDid,
                    counterpartyDid = insuredDid
                ),
                terms = ContractTerms(
                    obligations = listOf(
                        Obligation(
                            id = "payout-obligation",
                            partyDid = insurerDid,
                            description = "Pay out based on flood depth tier",
                            obligationType = ObligationType.PAYMENT
                        )
                    ),
                    conditions = listOf(
                        ContractCondition(
                            id = "flood-threshold-20cm",
                            description = "Flood depth >= 20cm (Tier 1)",
                            conditionType = ConditionType.THRESHOLD,
                            expression = "$.floodDepthCm >= 20"
                        ),
                        ContractCondition(
                            id = "flood-threshold-50cm",
                            description = "Flood depth >= 50cm (Tier 2)",
                            conditionType = ConditionType.THRESHOLD,
                            expression = "$.floodDepthCm >= 50"
                        ),
                        ContractCondition(
                            id = "flood-threshold-100cm",
                            description = "Flood depth >= 100cm (Tier 3)",
                            conditionType = ConditionType.THRESHOLD,
                            expression = "$.floodDepthCm >= 100"
                        )
                    ),
                    jurisdiction = "US",
                    governingLaw = "State of North Carolina"
                ),
                effectiveDate = Instant.now().toString(),
                expirationDate = Instant.now().plusSeconds(365 * 24 * 60 *
60).toString(),
                contractData = buildJsonObject {
                    put("productType", "SarFlood")
                    put("coverageAmount", coverageAmount)
                    put("location", buildJsonObject {
                        put("latitude", location.latitude)
                        put("longitude", location.longitude)
                        put("address", location.address)
                        put("region", location.region)
                    })
                    put("thresholds", buildJsonObject {
                        put("tier1Cm", 20.0)
                        put("tier2Cm", 50.0)
                        put("tier3Cm", 100.0)
                    })
                    put("payoutTiers", buildJsonObject {
                        put("tier1", 0.25)  // 25% of coverage
                        put("tier2", 0.50)  // 50% of coverage
                        put("tier3", 1.0)   // 100% of coverage
                    })
                }
            )
        ).getOrThrow()

        println("✅ Flood contract draft created: ${contract.id}")
        return contract
    }

    /**
     * Bind contract (issue VC and anchor to blockchain)
     */
    suspend fun bindFloodContract(
        contract: SmartContract,
        insurerDid: String,
        insurerKeyId: String
    ): BoundContract {

        val bound = vericore.contracts.bindContract(
            contractId = contract.id,
            issuerDid = insurerDid,
            issuerKeyId = insurerKeyId,
            chainId = "algorand:mainnet"
        ).getOrThrow()

        println("✅ Contract bound: ${bound.credentialId}, anchored:
```

```
${bound.anchorRef.txHash}")
        return bound
    }

    /**
     * Process SAR flood data and issue verifiable credential
     */
    suspend fun processSarFloodData(
        location: Location,
        sarData: SarFloodMeasurement,
        timestamp: Instant
    ): Result<VerifiableCredential> = vericoreCatching {

        // Step 1: Create EO data payload
        val floodData = buildJsonObject {
            put("id", "sar-flood-${location.id}-${timestamp.toEpochMilli()}")
            put("type", "SarFloodMeasurement")
            put("location", buildJsonObject {
                put("latitude", location.latitude)
                put("longitude", location.longitude)
                put("address", location.address)
                put("region", location.region)
            })
            put("measurement", buildJsonObject {
                put("floodDepthCm", sarData.floodDepthCm)
                put("floodAreaSqKm", sarData.floodAreaSqKm)
                put("confidence", sarData.confidence)
                put("source", "Sentinel-1 SAR")
                put("processingMethod", "SAR Flood Extraction")
                put("demUsed", sarData.demUsed)
                put("timestamp", timestamp.toString())
            })
            put("quality", buildJsonObject {
                put("validationStatus", "validated")
                put("dataQuality", sarData.quality)
            })
        }

        // Step 2: Compute data digest for integrity
        val dataDigest = DigestUtils.sha256DigestMultibase(floodData)

        // Step 3: Issue verifiable credential for EO data
        val eoProviderKeyId = vericore.dids.resolve(eoProviderDid)
            .getOrThrow()
            .verificationMethod
            .firstOrNull()?.id
            ?: error("No verification method found")

        val floodCredential = vericore.credentials.issue(
            issuer = eoProviderDid,
            subject = buildJsonObject {
                put("id", "sar-flood-${location.id}-${timestamp.toEpochMilli()}")
                put("dataType", "SarFloodMeasurement")
                put("data", floodData)
                put("dataDigest", dataDigest)
                put("provider", eoProviderDid)
                put("timestamp", timestamp.toString())
            },
            config = IssuanceConfig(
                proofType = ProofType.Ed25519Signature2020,
                keyId = eoProviderKeyId
            ),
            types = listOf(
                "VerifiableCredential",
                "EarthObservationCredential",
                "InsuranceOracleCredential",
                "SarFloodCredential"
            )
        ).getOrThrow()

        // Step 4: Anchor to blockchain for tamper-proof record
        val anchorResult = vericore.blockchains.anchor(
            data = floodCredential,
            serializer = VerifiableCredential.serializer(),
            chainId = "algorand:mainnet"
        ).getOrThrow()

        println("✅ SAR Flood Credential issued and anchored: 
${anchorResult.ref.txHash}")

        floodCredential
    }

    /**
     * Execute contract with flood data
     */
    suspend fun executeFloodContract(
        contract: SmartContract,
        floodCredential: VerifiableCredential
    ): ExecutionResult {

        // Extract flood depth from credential
        val credentialSubject = floodCredential.credentialSubject
        val floodDepthCm = credentialSubject.jsonObject["data"]
            ?.jsonObject?.get("measurement")
            ?.jsonObject?.get("floodDepthCm")
            ?.jsonPrimitive?.content?.toDouble()
            ?: error("Flood depth not found in credential")

        // Create execution context with trigger data
```

```kotlin
        val executionContext = ExecutionContext(
            triggerData = buildJsonObject {
                put("floodDepthCm", floodDepthCm)
                put("credentialId", floodCredential.id)
                put("timestamp", Instant.now().toString())
            }
        )

        // Execute contract
        val result = vericore.contracts.executeContract(
            contract = contract,
            executionContext = executionContext
        ).getOrThrow()

        if (result.executed) {
            println("✅ Contract executed! Payout triggered for flood depth: ${floodDepthCm}cm")
            result.outcomes.forEach { outcome ->
                println("   Outcome: ${outcome.description}")
                outcome.monetaryImpact?.let {
                    println("   Amount: ${it.amount} ${it.currency}")
                }
            }
        } else {
            println("⚠️  Contract conditions not met (flood depth: ${floodDepthCm}cm)")
        }

        return result
    }
}

// Data Models
data class Location(
    val id: String,
    val latitude: Double,
    val longitude: Double,
    val address: String,
    val region: String
)

data class SarFloodMeasurement(
    val floodDepthCm: Double,
    val floodAreaSqKm: Double,
    val confidence: Double,
    val demUsed: String,
    val quality: String
)
```

Product 2: Heatwave Parametric

Data Sources: MODIS LST + ERA5 **Markets:** GCC (Saudi Arabia, UAE) **Triggers:** > X°C for Y days **Clients:** Construction, energy, government

Implementation

```
package com.atlasparametric.products.heatwave

import com.geoknoesis.vericore.VeriCore
import com.geoknoesis.vericore.core.*
import com.geoknoesis.vericore.json.DigestUtils
import kotlinx.serialization.json.buildJsonObject
import kotlinx.serialization.json.put
import java.time.Instant
import java.time.Duration

class HeatwaveProduct(
    private val vericore: VeriCore,
    private val eoProviderDid: String
) {

    /**
     * Process heatwave data from MODIS LST
     */
    suspend fun processHeatwaveData(
        location: Location,
        lstData: List<LstMeasurement>,
        threshold: HeatwaveThreshold
    ): Result<VerifiableCredential> = vericoreCatching {

        // Calculate consecutive days above threshold
        val consecutiveDays = calculateConsecutiveDays(lstData,
threshold.temperatureC)

        val heatwaveData = buildJsonObject {
            put("id", "heatwave-${location.id}-${Instant.now().toEpochMilli()}")
            put("type", "HeatwaveMeasurement")
            put("location", buildJsonObject {
                put("latitude", location.latitude)
                put("longitude", location.longitude)
                put("region", location.region)
            })
            put("measurement", buildJsonObject {
                put("maxTemperatureC", lstData.maxOf { it.temperatureC })
                put("avgTemperatureC", lstData.average { it.temperatureC })
                put("consecutiveDays", consecutiveDays)
                put("thresholdC", threshold.temperatureC)
                put("minDaysRequired", threshold.minDays)
                put("source", "MODIS LST + ERA5")
                put("timestamp", Instant.now().toString())
            })
        }

        val dataDigest = DigestUtils.sha256DigestMultibase(heatwaveData)

        val eoProviderKeyId = vericore.dids.resolve(eoProviderDid)
            .getOrThrow()
            .verificationMethod
            .firstOrNull()?.id
            ?: error("No verification method found")

        val heatwaveCredential = vericore.credentials.issue(
            issuerDid = eoProviderDid,
            issuerKeyId = eoProviderKeyId,
            credentialSubject = buildJsonObject {
                put("id", "heatwave-${location.id}-${Instant.now().toEpochMilli()}")
                put("dataType", "HeatwaveMeasurement")
                put("data", heatwaveData)
                put("dataDigest", dataDigest)
                put("provider", eoProviderDid)
                put("timestamp", Instant.now().toString())
            },
            types = listOf(
                "VerifiableCredential",
                "EarthObservationCredential",
                "InsuranceOracleCredential",
                "HeatwaveCredential"
            )
        ).getOrThrow()

        // Anchor to blockchain
        vericore.blockchains.anchor(
            data = heatwaveCredential,
            serializer = VerifiableCredential.serializer(),
            chainId = "algorand:mainnet"
        ).getOrThrow()

        heatwaveCredential
    }

    /**
     * Create heatwave insurance contract
     */
    suspend fun createHeatwaveContract(
        insurerDid: String,
        insuredDid: String,
        basePayout: Double,
        threshold: HeatwaveThreshold,
        location: Location
    ): SmartContract {

        val contract = vericore.contracts.draft(
            request = ContractDraftRequest(
                contractType = ContractType.Insurance,
                executionModel = ExecutionModel.Parametric(
                    triggerType = TriggerType.EarthObservation,
                    evaluationEngine = "parametric-insurance"
                ),
                parties = ContractParties(
                    primaryPartyDid = insurerDid,
                    counterpartyDid = insuredDid
                ),
                terms = ContractTerms(
                    obligations = listOf(
                        Obligation(
                            id = "heatwave-payout",
                            partyDid = insurerDid,
                            description = "Pay out for consecutive days above
threshold",
                            obligationType = ObligationType.PAYMENT
                        )
                    ),
                    conditions = listOf(
                        ContractCondition(
                            id = "heatwave-threshold",
                            description = "Temperature >= ${threshold.temperatureC}
°C for ${threshold.minDays} days",
                            conditionType = ConditionType.COMPOSITE,
                            expression = "$.consecutiveDays >= ${threshold.minDays}
&& $.maxTemperatureC >= ${threshold.temperatureC}"
                        )
                    )
                ),
                effectiveDate = Instant.now().toString(),
                expirationDate = Instant.now().plusSeconds(365 * 24 * 60 *
60).toString(),
                contractData = buildJsonObject {
                    put("productType", "Heatwave")
                    put("basePayout", basePayout)
                    put("threshold", buildJsonObject {
                        put("temperatureC", threshold.temperatureC)
                        put("minDays", threshold.minDays)
                    })
                    put("location", buildJsonObject {
                        put("latitude", location.latitude)
                        put("longitude", location.longitude)
                        put("region", location.region)
```

```kotlin
                })
            }
        )
    ).getOrThrow()

    return contract
}

/**
 * Execute heatwave contract
 */
suspend fun executeHeatwaveContract(
    contract: SmartContract,
    heatwaveCredential: VerifiableCredential
): ExecutionResult {

    val credentialSubject = heatwaveCredential.credentialSubject
    val consecutiveDays = credentialSubject.jsonObject["data"]
        ?.jsonObject?.get("measurement")
        ?.jsonObject?.get("consecutiveDays")
        ?.jsonPrimitive?.content?.toInt()
        ?: error("Consecutive days not found")

    val maxTemp = credentialSubject.jsonObject["data"]
        ?.jsonObject?.get("measurement")
        ?.jsonObject?.get("maxTemperatureC")
        ?.jsonPrimitive?.content?.toDouble()
        ?: error("Max temperature not found")

    val executionContext = ExecutionContext(
        triggerData = buildJsonObject {
            put("consecutiveDays", consecutiveDays)
            put("maxTemperatureC", maxTemp)
            put("credentialId", heatwaveCredential.id)
        }
    )

    return vericore.contracts.executeContract(
        contract = contract,
        executionContext = executionContext
    ).getOrThrow()
}

private fun calculateConsecutiveDays(
    lstData: List<LstMeasurement>,
    thresholdC: Double
): Int {
    var maxConsecutive = 0
    var currentConsecutive = 0
```

```
        for (measurement in lstData.sortedBy { it.timestamp }) {
            if (measurement.temperatureC > thresholdC) {
                currentConsecutive++
                maxConsecutive = maxOf(maxConsecutive, currentConsecutive)
            } else {
                currentConsecutive = 0
            }
        }

        return maxConsecutive
    }
}

data class LstMeasurement(
    val temperatureC: Double,
    val timestamp: Instant
)

data class HeatwaveThreshold(
    val temperatureC: Double,
    val minDays: Int
)

data class HeatwavePolicy(
    val id: String,
    val location: Location,
    val threshold: HeatwaveThreshold,
    val basePayout: Double,
    val dailyIncrement: Double
)
```

Product 3: Solar Attenuation Parametric

Data Sources: AOD (MODIS, VIIRS) + Irradiance (CERES) **Markets:** GCC (Solar farms) **Triggers:** >30% irradiance drop **Clients:** ACWA Power, NEOM, UAE utilities

Implementation

```
package com.atlasparametric.products.solar

import com.geoknoesis.vericore.VeriCore
import com.geoknoesis.vericore.core.*
import com.geoknoesis.vericore.json.DigestUtils
import kotlinx.serialization.json.buildJsonObject
import kotlinx.serialization.json.put
import java.time.Instant
```

```
class SolarAttenuationProduct(
    private val vericore: VeriCore,
    private val eoProviderDid: String
) {

    /**
     * Process solar attenuation data
     */
    suspend fun processSolarAttenuation(
        location: Location,
        aodData: AodMeasurement,
        irradianceData: IrradianceMeasurement
    ): Result<VerifiableCredential> = vericoreCatching {

        // Calculate attenuation percentage
        val baselineIrradiance = irradianceData.baselineWattPerSqM
        val currentIrradiance = irradianceData.currentWattPerSqM
        val attenuationPercent = ((baselineIrradiance - currentIrradiance) /
baselineIrradiance) * 100.0

        val solarData = buildJsonObject {
            put("id", "solar-
attenuation-${location.id}-${Instant.now().toEpochMilli()}")
            put("type", "SolarAttenuationMeasurement")
            put("location", buildJsonObject {
                put("latitude", location.latitude)
                put("longitude", location.longitude)
                put("solarFarmId", location.solarFarmId)
            })
            put("measurement", buildJsonObject {
                put("aod", aodData.aodValue)
                put("baselineIrradiance", baselineIrradiance)
                put("currentIrradiance", currentIrradiance)
                put("attenuationPercent", attenuationPercent)
                put("source", "MODIS/VIIRS AOD + CERES Irradiance")
                put("timestamp", Instant.now().toString())
            })
        }

        val dataDigest = DigestUtils.sha256DigestMultibase(solarData)

        val eoProviderKeyId = vericore.dids.resolve(eoProviderDid)
            .getOrThrow()
            .verificationMethod
            .firstOrNull()?.id
            ?: error("No verification method found")

        val solarCredential = vericore.credentials.issue(
            issuerDid = eoProviderDid,
            issuerKeyId = eoProviderKeyId,
            credentialSubject = buildJsonObject {
                put("id", "solar-
attenuation-${location.id}-${Instant.now().toEpochMilli()}")
                put("dataType", "SolarAttenuationMeasurement")
                put("data", solarData)
                put("dataDigest", dataDigest)
                put("provider", eoProviderDid)
                put("timestamp", Instant.now().toString())
            },
            types = listOf(
                "VerifiableCredential",
                "EarthObservationCredential",
                "InsuranceOracleCredential",
                "SolarAttenuationCredential"
            )
        ).getOrThrow()

        // Anchor to blockchain
        vericore.blockchains.anchor(
            data = solarCredential,
            serializer = VerifiableCredential.serializer(),
            chainId = "algorand:mainnet"
        ).getOrThrow()

        solarCredential
    }

    /**
     * Evaluate solar attenuation trigger
     */
    suspend fun evaluateSolarTrigger(
        policy: SolarPolicy,
        solarCredential: VerifiableCredential
    ): TriggerResult {

        val verification = vericore.credentials.verify(solarCredential)
        if (!verification.valid) {
            return TriggerResult(triggered = false, reason = "Credential invalid")
        }

        val credentialSubject = solarCredential.credentialSubject
        val attenuationPercent = credentialSubject.jsonObject["data"]
            ?.jsonObject?.get("measurement")
            ?.jsonObject?.get("attenuationPercent")
            ?.jsonPrimitive?.content?.toDouble()
            ?: return TriggerResult(triggered = false, reason = "Attenuation not
found")

        val thresholdPercent = policy.thresholdPercent
        if (attenuationPercent < thresholdPercent) {
            return TriggerResult(
                triggered = false,
                reason = "Attenuation ($attenuationPercent%) below threshold
($thresholdPercent%)"
            )
        }

        // Calculate payout based on attenuation severity
        val excessAttenuation = attenuationPercent - thresholdPercent
        val payoutAmount = when {
            excessAttenuation >= 20 -> policy.coverageAmount * 1.0  // Full payout
            excessAttenuation >= 10 -> policy.coverageAmount * 0.75
            else -> policy.coverageAmount * 0.50
        }

        return TriggerResult(
            triggered = true,
            attenuationPercent = attenuationPercent,
            payoutAmount = payoutAmount,
            dataCredentialId = solarCredential.id
        )
    }
}

data class AodMeasurement(
    val aodValue: Double,
    val timestamp: Instant
)

data class IrradianceMeasurement(
    val baselineWattPerSqM: Double,
    val currentWattPerSqM: Double,
    val timestamp: Instant
)

data class SolarPolicy(
    val id: String,
    val location: Location,
    val thresholdPercent: Double = 30.0,
    val coverageAmount: Double
)
```

Complete Workflow Example

Complete Flood Insurance Workflow with Smart Contracts

```
suspend fun completeFloodInsuranceWorkflow() {
    // Step 1: Initialize VeriCore
    val vericore = VeriCore.create {
        blockchains {
            "algorand:mainnet" to algorandClient
        }
    }

    // Step 2: Create DIDs for parties
    val insurerDid = vericore.dids.create(method = "key")
    val insuredDid = vericore.dids.create(method = "key")
    val eoProviderDid = vericore.dids.create(method = "key")
    val insurerKeyId = vericore.dids.resolve(insurerDid.id)
        .verificationMethod.firstOrNull()?.id ?: error("No key found")

    // Step 3: Initialize product
    val floodProduct = SarFloodProduct(vericore, eoProviderDid.id)

    // Step 4: Create contract
    val contract = floodProduct.createFloodContract(
        insurerDid = insurerDid.id,
        insuredDid = insuredDid.id,
        coverageAmount = 1_000_000.0,
        location = Location(
            id = "loc-001",
            latitude = 35.2271,
            longitude = -80.8431,
            address = "Charlotte, NC",
            region = "North Carolina"
        )
    )

    // Step 5: Bind contract (issue VC and anchor)
    val bound = floodProduct.bindFloodContract(
        contract = contract,
        insurerDid = insurerDid.id,
        insurerKeyId = insurerKeyId
    )

    // Step 6: Activate contract
    val active = vericore.contracts.activateContract(bound.contract.id).getOrThrow()

    // Step 7: Process EO data (in production, this comes from EO provider)
    val floodCredential = floodProduct.processSarFloodData(
        location = Location(
            id = "loc-001",
            latitude = 35.2271,
            longitude = -80.8431,
            address = "Charlotte, NC",
            region = "North Carolina"
        ),
        sarData = SarFloodMeasurement(
            floodDepthCm = 75.0,
            floodAreaSqKm = 15.5,
            confidence = 0.95,
            demUsed = "SRTM",
            quality = "high"
        ),
        timestamp = Instant.now()
    ).getOrThrow()

    // Step 8: Execute contract
    val executionResult = floodProduct.executeFloodContract(
        contract = active,
        floodCredential = floodCredential
    )

    // Step 9: Process payout (application-specific)
    if (executionResult.executed) {
        processPayout(executionResult)
    }
}
```

Complete System Integration

Main Application

```
package com.atlasparametric

import com.geoknoesis.vericore.VeriCore
import com.geoknoesis.vericore.chains.algorand.AlgorandBlockchainAnchorClient
import com.atlasparametric.products.flood.SarFloodProduct
import com.atlasparametric.products.heatwave.HeatwaveProduct
import com.atlasparametric.products.solar.SolarAttenuationProduct
import kotlinx.coroutines.runBlocking

/**
 * Atlas Parametric MGA Platform
 *
 * Main application entry point
 */
class AtlasParametricPlatform {

    private val vericore: VeriCore
    private val sarFloodProduct: SarFloodProduct
    private val heatwaveProduct: HeatwaveProduct
    private val solarProduct: SolarAttenuationProduct
    init {
        // Initialize VeriCore with blockchain anchoring
        vericore = VeriCore.create {
            blockchains {
                "algorand:mainnet" to AlgorandBlockchainAnchorClient(
                    chainId = "algorand:mainnet",
                    apiKey = System.getenv("ALGORAND_API_KEY")
                )
            }
        }

        // Create DIDs for EO providers
        val eoProviderDid = runBlocking {
            vericore.dids.create(method = "key")
        }

        // Initialize products
        sarFloodProduct = SarFloodProduct(vericore, eoProviderDid.id)
        heatwaveProduct = HeatwaveProduct(vericore, eoProviderDid.id)
        solarProduct = SolarAttenuationProduct(vericore, eoProviderDid.id)
    }

    /**
     * Process EO data and execute contracts
     */
    suspend fun processEoDataAndExecute(
        productType: ProductType,
        eoData: Any,
        contracts: List<SmartContract>
    ): List<ExecutionResult> {

        return when (productType) {
            ProductType.SAR_FLOOD -> {
                val floodData = eoData as SarFloodMeasurement
                processFloodExecutions(floodData, contracts)
            }
            ProductType.HEATWAVE -> {
                val heatData = eoData as List<LstMeasurement>
                processHeatwaveExecutions(heatData, contracts)
            }
            ProductType.SOLAR_ATTENUATION -> {
                val solarData = eoData as Pair<AodMeasurement,
IrradianceMeasurement>
                processSolarExecutions(solarData, contracts)
            }
        }
    }

    private suspend fun processFloodExecutions(
        floodData: SarFloodMeasurement,
        contracts: List<SmartContract>
    ): List<ExecutionResult> {

        val results = mutableListOf<ExecutionResult>()

        for (contract in contracts.filter {
            it.contractData.jsonObject["productType"]?.jsonPrimitive?.content ==
"SarFlood"
        }) {
            // Process SAR data and issue credential
            val location = extractLocation(contract)
            val floodCredential = sarFloodProduct.processSarFloodData(
                location = location,
                sarData = floodData,
                timestamp = Instant.now()
            ).getOrThrow()

            // Execute contract
            val executionResult = sarFloodProduct.executeFloodContract(
                contract = contract,
                floodCredential = floodCredential
            )

            if (executionResult.executed) {
                // Process payout
                processPayout(executionResult)
            }

            results.add(executionResult)
        }

        return results
    }

    private fun extractLocation(contract: SmartContract): Location {
        val locationData = contract.contractData.jsonObject["location"]?.jsonObject
            ?: error("Location not found in contract")

        return Location(
            id = locationData["address"]?.jsonPrimitive?.content ?: "unknown",
            latitude = locationData["latitude"]?.jsonPrimitive?.content?.toDouble()
?: 0.0,
            longitude =
locationData["longitude"]?.jsonPrimitive?.content?.toDouble() ?: 0.0,
            address = locationData["address"]?.jsonPrimitive?.content ?: "",
            region = locationData["region"]?.jsonPrimitive?.content ?: ""
        )
    }
    private suspend fun processPayout(executionResult: ExecutionResult) {
        // Integrate with banking API (Stripe, Plaid, etc.)
        // Extract payout amount from executionResult.outcomes
        executionResult.outcomes.forEach { outcome ->
            outcome.monetaryImpact?.let { amount ->
                // Execute payout via banking API
                println("Processing payout: ${amount.amount} ${amount.currency}")
            }
        }
    }
}

enum class ProductType {
    SAR_FLOOD,
    HEATWAVE,
    SOLAR_ATTENUATION,
    HURRICANE,
    DROUGHT
}

data class PayoutResult(
    val policyId: String,
    val amount: Double,
    val status: String,
    val payoutCredentialId: String,
    val timestamp: Instant
)
```

Multi-Provider EO Data Acceptance

One of VeriCore's key advantages is accepting EO data from multiple providers:

```
/**
 * Accept EO data from any certified provider
 */
class MultiProviderEoDataService(
    private val vericore: VeriCore
) {

    private val certifiedProviders = setOf(
        "did:key:esa-provider",
        "did:key:planet-provider",
        "did:key:nasa-provider",
        "did:key:noaa-provider"
    )

    /**
     * Accept EO data credential from any certified provider
     */
    suspend fun acceptEoDataCredential(
        dataCredential: VerifiableCredential
    ): Result<EoData> = vericoreCatching {

        // Step 1: Verify credential
        val verification = vericore.verifyCredential(dataCredential).getOrThrow()
        if (!verification.valid) {
            error("Credential verification failed: ${verification.errors}")
        }

        // Step 2: Check if provider is certified
        val issuerDid = dataCredential.issuer.firstOrNull()?.id
            ?: error("No issuer found in credential")

        if (!certifiedProviders.contains(issuerDid)) {
            error("Provider $issuerDid is not certified")
        }

        // Step 3: Extract and return data
        val credentialSubject = dataCredential.credentialSubject
        val dataType =
credentialSubject.jsonObject["dataType"]?.jsonPrimitive?.content
        val data = credentialSubject.jsonObject["data"]?.jsonObject

        EoData(
            type = dataType ?: "Unknown",
            data = data ?: buildJsonObject {},
            provider = issuerDid,
            credentialId = dataCredential.id
        )
    }
}

data class EoData(
    val type: String,
    val data: JsonObject,
    val provider: String,
    val credentialId: String
)
```

Broker Portal Integration

```
/**
 * Broker Portal API
 */
@RestController
@RequestMapping("/api/broker")
```

```
class BrokerPortalController(
    private val platform: AtlasParametricPlatform,
    private val pricingEngine: PricingEngine
) {

    /**
     * Get quote for parametric insurance
     */
    @PostMapping("/quote")
    suspend fun getQuote(
        @RequestBody request: QuoteRequest
    ): QuoteResponse {

        val premium = pricingEngine.calculatePremium(
            productType = request.productType,
            location = request.location,
            coverageAmount = request.coverageAmount
        )

        return QuoteResponse(
            premium = premium,
            coverageAmount = request.coverageAmount,
            productType = request.productType
        )
    }

    /**
     * Bind policy
     */
    @PostMapping("/bind")
    suspend fun bindPolicy(
        @RequestBody request: BindRequest
    ): PolicyResponse {

        // Create policy
        val policy = createPolicy(request)

        // Issue policy credential
        val policyCredential = issuePolicyCredential(policy)

        return PolicyResponse(
            policyId = policy.id,
            policyCredentialId = policyCredential.id,
            status = "BOUND"
        )
    }
}
```

Regulatory Compliance & Audit Trails

```
/**
 * Audit Trail Service using VeriCore blockchain anchoring
 */
class AuditTrailService(
    private val vericore: VeriCore
) {

    /**
     * Record audit event and anchor to blockchain
     */
    suspend fun recordAuditEvent(
        event: AuditEvent
    ): AuditRecord {

        // Create audit record
        val auditRecord = buildJsonObject {
            put("id", event.id)
            put("timestamp", event.timestamp.toString())
            put("eventType", event.type)
            put("actor", event.actor)
            put("resource", event.resource)
            put("details", event.details)
        }

        // Anchor to blockchain for immutability
        val anchorResult = vericore.blockchains.anchor(
            data = auditRecord,
            serializer = JsonObject.serializer(),
            chainId = "algorand:mainnet"
        ).getOrThrow()

        return AuditRecord(
            eventId = event.id,
            anchorRef = anchorResult.ref,
            timestamp = event.timestamp
        )
    }

    /**
     * Verify audit record integrity
     */
    suspend fun verifyAuditRecord(
        record: AuditRecord
    ): Boolean {

        // Read from blockchain
        val client = vericore.getBlockchainClient(record.anchorRef.chainId)
            ?: return false
        val anchorResult = client.readPayload(record.anchorRef)

        // Verify integrity
        return anchorResult.payload.jsonObject["id"]?.jsonPrimitive?.content ==
record.eventId
    }
}
```

Deployment Strategy

Phase 1: MVP (Weeks 1-6)

1. **Setup VeriCore**
    - Initialize VeriCore with Algorand or Polygon
    - Create DIDs for EO providers
    - Setup blockchain anchoring
2. **Build SAR Flood Product**
    - Implement SAR flood detection
    - Create trigger evaluation logic
    - Build payout automation
3. **Broker Portal MVP**
    - Quote generation
    - Policy binding
    - Trigger dashboard

Phase 2: Production (Months 2-12)

1. **Add Heatwave Product**
2. **Add Solar Attenuation Product**
3. **Multi-provider EO data acceptance**
4. **Regulatory compliance features**
5. **Reinsurer dashboard**

Key Benefits of Using VeriCore

1. **Standardization**: W3C-compliant format works with all EO providers
2. **Trust**: Cryptographic proof of data integrity
3. **Multi-Provider**: Accept data from ESA, Planet, NASA without custom integrations
4. **Regulatory Compliance**: Blockchain-anchored audit trails
5. **Automation**: Enable instant payouts with verifiable triggers
6. **Cost Reduction**: Eliminate custom API integrations

Next Steps

1. **Review Existing Scenarios**:
   - Parametric Insurance with EO Data
   - Earth Observation Scenario
2. **Explore VeriCore APIs**:
   - Core API Reference
   - Blockchain Anchoring
3. **Start Building**:
   - Clone VeriCore repository
   - Follow Quick Start Guide
   - Implement SAR flood product first

Related Documentation

- Parametric Insurance with EO Data - EO data insurance patterns
- Earth Observation Scenario - EO data integrity
- Blockchain Anchoring - Anchoring concepts
- API Reference - Complete API documentation