

# Atlas Parametric MGA: Quick Reference

## How to Build Your Parametric Insurance MGA with VeriCore

### ⌚ What You're Building

A parametric insurance MGA platform that:

- Uses EO data (SAR, NDVI, AOD, LST) for objective triggers
- Pays out automatically in 24-72 hours
- Accepts EO data from multiple providers (ESA, Planet, NASA, NOAA)
- Provides tamper-proof audit trails via blockchain
- Scales globally with standardized format

### 🏗 Architecture in 4 Steps

#### Step 1: Create Smart Contract

```
// Create parametric insurance contract
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(...),
        effectiveDate = Instant.now().toString(),
        contractData = buildJsonObject { ... }
    )
).getOrThrow()
```

#### Step 2: Bind Contract (Issue VC & Anchor)

```
// Bind contract - issues VC and anchors to blockchain
val bound = vericore.contracts.bindContract(
    contractId = contract.id,
    issuerDid = insurerDid,
    issuerKeyId = insurerKeyId,
```

```

chainId = "algorand:mainnet"
).getOrThrow()

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

```

### Step 3: EO Provider Issues Data Credential

```

// EO provider wraps SAR flood data in Verifiable Credential
val floodCredential = vericore.credentials.issue(
    issuer = eoProviderDid,
    subject = sarFloodData,
    types = listOf("EarthObservationCredential", "InsuranceOracleCredential")
).getOrThrow()

```

### Step 4: Execute Contract & Payout

```

// Execute contract with EO data
val result = vericore.contracts.executeContract(
    contract = active,
    executionContext = ExecutionContext(
        triggerData = buildJsonObject {
            put("floodDepthCm", 75.0)
            put("credentialId", floodCredential.id)
        }
    )
).getOrThrow()

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

```

## 📦 Products You Can Build

### 1. SAR Flood Parametric

- **Data:** Sentinel-1 SAR + DEM
- **Trigger:** Flood depth thresholds (20cm, 50cm, 1m)
- **Market:** US (NC, SC, FL, GA)
- **Payout:** \$25k - \$5M

### 2. Heatwave Parametric

- **Data:** MODIS LST + ERA5
- **Trigger:** > X°C for Y consecutive days
- **Market:** GCC (Saudi Arabia, UAE)
- **Clients:** Construction, energy, government

### 3. Solar Attenuation Parametric

- **Data:** AOD (MODIS/VIIRS) + Irradiance (CERES)
- **Trigger:** >30% irradiance drop
- **Market:** GCC solar farms
- **Clients:** ACWA Power, NEOM, UAE utilities

### 4. Hurricane Parametric

- **Data:** NOAA EO + microwave satellites
- **Trigger:** Wind speed + rainfall indices
- **Market:** Caribbean, US Gulf Coast

### 5. Drought/Agriculture Parametric

- **Data:** NDVI (Sentinel-2, MODIS) + soil moisture
- **Trigger:** NDVI anomaly + rainfall deficit
- **Market:** Africa, Asia

## Key VeriCore Features

| Feature                       | What It Does  | Why It Matters                                |
|-------------------------------|---|---|
| <b>DID Management</b>         | Creates identities for EO providers, insurers, reinsurers | Standardized identity across ecosystem        |
| <b>Verifiable Credentials</b> | Wraps EO data with cryptographic proof                    | Prevents tampering, enables trust             |
| <b>Blockchain Anchoring</b>   | Creates tamper-proof audit trails                         | Regulatory compliance, dispute resolution     |
| <b>Multi-Provider Support</b> | Accepts EO data from any certified provider               | No custom integrations needed                 |
| <b>Data Integrity</b>         | Cryptographic digests verify data hasn't changed          | Prevents replay attacks, ensures authenticity |

## Business Value

### Cost Savings

- **80% reduction** in integration costs (no custom APIs per provider)

- **Standardized format** works with all EO providers
- **Automated verification** reduces manual review

## Speed to Market

- **Launch products faster** with standardized format
- **Add new providers** without code changes
- **Instant verification** enables 24-72 hour payouts

## Regulatory Compliance

- **Blockchain-anchored audit trails** for regulators
- **Cryptographic proof** of data integrity
- **Complete data lineage** for compliance

## Competitive Advantage

- **Only EO-first MGA** with full-spectrum EO integration
- **Multi-provider support** without vendor lock-in
- **Instant verification** vs. months for traditional insurance

## Implementation Phases

### Phase 1: MVP (Weeks 1-6)

- Setup VeriCore with blockchain anchoring
- Build SAR flood product
- Create broker portal MVP
- Implement trigger evaluation

### Phase 2: Production (Months 2-12)

- Add heatwave product
- Add solar attenuation product
- Multi-provider EO data acceptance
- Regulatory compliance features

### Phase 3: Scale (Months 12-24)

- Hurricane product
- Drought/NDVI product
- Enterprise licensing
- Global expansion

## Documentation

### Full Guides

1. [Implementation Guide](#) - Complete code examples
2. [Architecture Overview](#) - System design
3. [EO Scenario](#) - EO data patterns

## VeriCore Docs

- [Quick Start](#)
- [API Reference](#)
- [Blockchain Anchoring](#)

## Code Snippets

### Initialize VeriCore

```
val vericore = VeriCore.create {
    blockchains {
        "algorand:mainnet" to AlgorandBlockchainAnchorClient(...)
    }
}
```

### Create Flood Insurance Contract

```
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",

```

```

        conditionType = ConditionType.THRESHOLD,
        expression = "$.floodDepthCm >= 20"
    )
)
),
effectiveDate = Instant.now().toString(),
contractData = buildJsonObject {
    put("productType", "SarFlood")
    put("coverageAmount", 1_000_000.0)
}
)
).getOrThrow()

```

## Bind and Activate Contract

```

// Bind contract (issues VC and anchors)
val bound = vericore.contracts.bindContract(
    contractId = contract.id,
    issuerDid = insurerDid,
    issuerKeyId = insurerKeyId
).getOrThrow()

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

```

## Issue EO Data Credential

```

val floodCredential = vericore.credentials.issue(
    issuer = eoProviderDid,
    subject = buildJsonObject {
        put("dataType", "SarFloodMeasurement")
        put("data", floodData)
        put("dataDigest", dataDigest)
    },
    types = listOf("EarthObservationCredential", "InsuranceOracleCredential")
).getOrThrow()

```

## Execute Contract

```

val result = vericore.contracts.executeContract(
    contract = active,
    executionContext = ExecutionContext(
        triggerData = buildJsonObject {

```

```

        put("floodDepthCm", 75.0)
        put("credentialId", floodCredential.id)
    }
)
).getOrThrow()

if (result.executed) {
    // Process payout
    result.outcomes.forEach { outcome ->
        outcome.monetaryImpact?.let { amount ->
            println("Payout: ${amount.amount} ${amount.currency}")
        }
    }
}

```

## Accept Multi-Provider Data

```

// Works with ESA, Planet, NASA, NOAA - same format!
val eoData = acceptEoDataCredential(dataCredential)
// No custom integration needed!

```

## ⌚ Key Differentiators

vs. Traditional Insurance

- **24-72 hour payouts** vs. months
- **Objective triggers** vs. adjuster disputes
- **No exclusions** vs. coverage gaps
- **Automated** vs. manual processing

vs. Competitors

- **EO-first design** vs. sensor-based
- **Multi-provider support** vs. vendor lock-in
- **Blockchain audit trail** vs. traditional records
- **Standardized format** vs. custom integrations

## 📊 Market Opportunity

- **TAM:** \$250B (flood, hurricane, drought, heatwave losses)
- **SAM:** \$29B (parametric insurance market by 2030)
- **SOM:** \$200M premium within 5 years

## 🔗 Next Steps

1. **Read Implementation Guide:** [parametric-insurance-mga-implementation-guide.md](#)
  2. **Review Architecture:** [atlas-parametric-architecture-overview.md](#)
  3. **Smart Contracts Guide:** [Smart Contract: Parametric Insurance](#)
  4. **Explore VeriCore:** [Quick Start](#)
  5. **Start Building:** Begin with SAR flood product MVP using Smart Contracts
- 

**Built with VeriCore** - The Foundation for Decentralized Trust and Identity