# Icon Payments Framework (IPF) Briefing Document

## 1. Executive Summary

The Icon Payments Framework (IPF) is a comprehensive, modular, and extensible platform designed for building high-performance payment solutions. At its core, IPF leverages a Domain Specific Language (DSL) called "flo-lang" for defining orchestrated payment flows, which are then transformed into executable Java code. The platform emphasizes resilience, scalability, and integration capabilities, offering robust features for handling various aspects of payment processing, from message validation and routing to persistence, error handling, and operational monitoring. IPF utilizes an event-sourced architecture, storing immutable events to provide a complete audit trail and enable efficient recovery.

## 2. Core Concepts and Architecture

IPF's architecture is built around several key concepts:

• **Flo-lang (DSL)**: "this is the tool designed by IPF to enable users to create orchestrated payment flows." It allows users to visually design payment workflows, including states, events, decisions, and interactions with external systems.

• **Code Generation**: Flows defined in flo-lang are "transformed... into executable java code," ensuring consistency and reducing manual coding effort.

• **Event Sourcing**: IPF primarily uses event sourcing, where "only the events that are persisted by the actor are stored, not the actual state of the actor." This provides an immutable audit trail, crucial for financial transactions, and enables robust recovery by replaying events.

• **Akka Framework**: IPF is built on Akka, utilizing its actor model for "highly scalable, concurrent processing," and Akka Streams for managing data flow with "back-pressure mechanisms to handle varying latencies."

• **Modularity and Reuse**: IPF is designed for reuse, allowing "DSL components to be shared across different solutions, enabling quicker and faster implementations." This is evident in its modular structure with various starter modules and client libraries.

## 3. Key Components and Features

### 3.1. Orchestration Framework (Flo-lang)

The heart of IPF, the orchestration framework, enables the creation and management of payment flows.

• **Flow Definition**: A flow is uniquely identified by its "Flow Name" and "Flow Version" (e.g., "TestV3").

• **States and Events**: Flows transition between "States" based on "Events." This movement can be triggered "On any of" multiple events or "On all of" multiple events.

  ◦ **Domain Events**: "the type of event that is explicitly declared within any MPS solution."

  ◦ **System Events**: "occur when something happens to the system and can be tailored to be specific to individual needs."

  ◦ **Action Timeout Events**: "occur during processing when configured timeout settings are broken."

  ◦ **Decision Events**: "used as a response to receiving results from decisions."

• **Behaviors**:

  ◦ **Initiation Behaviour**: A special input behavior "only used to start a flow."

  ◦ **Input Behaviour**: Defines how the flow reacts to external "Instructions" (initiated by external systems, expecting no response) and "Requests" (expecting a "response" back). Responses can be "Completing" or "non-completing."

  ◦ **Event Behaviour**: Defines actions based on an event received in a "Current State."

• **Decisions**: Allow "conditional logic" (e.g., "only want to run a fraud check if the value of the payment is over £50"). Decisions have "Decision Outcomes" and are stored in a "Decision Library" for reuse.

• **Mapping Functions**: Enable "many-to-many translation of business data elements."
  ◦ **Aggregate Functions**: Used for calculations on "in flight" data not persisted, replayed during recovery.
  ◦ **Input Enrichment**: "generate (or update) business data elements to be stored on a received event" and are "persisted."
• **Flow-to-Flow Communication**: IPF supports calling other flows, either as "subflows" or separate, distinct flows. This enables breaking down complex processes into smaller, manageable units.
• **Timeouts and Retries**: IPF provides mechanisms to "add a timeout to your flow so that you can then perform custom compensation processing" when external components don't reply as expected. Automated retries are configurable with "initial-retry-interval," "max-retries," and "jitter-factor."
• **Persistence Purging**: Allows enabling "TTL purging" for journal and snapshot collections in MongoDB/CosmosDB to manage data retention.

### 3.2. Connectivity Components (Connectors)

The Connector Framework facilitates reliable integration with external systems.
• **Sending Connectors**: "responsible for taking messages supplied by a client and sending it over the configured transport mechanism."
• **Receiving Connectors**: Consume messages from external domains and convert them into internal Input objects.
• **HTTP Receive Flows**: Provide a standardized way to implement logging, correlation, and other features when exposing an HTTP endpoint via mechanisms other than a receive connector (e.g., Spring controller).
• **Supported Transports**: Kafka, JMS, and HTTP are natively supported. Custom transports can be developed.
• **Resilience**: Connectors incorporate "retries, circuit breaking and re-routing" using the Resilience4j library to cope with transient failures. Load-balancing across multiple transports is supported with various routing logic (default is round-robin).
• **Message Throttling**: Configurable on both sending and receiving connectors to "avoid overloading downstream systems and even out spiky traffic."
• **Message Encryption**: Connectors can be configured to "encrypt or decrypt messages when sending or receiving messages, respectively," adding application-level security.
• **Message Validation**: Allows providing a `Validator` implementation (e.g., for JSON Schema or XML Schema) to validate `TransportMessage` before sending or receiving.
• **Message Correlation**: Essential for asynchronous request-reply patterns. `CorrelationIdExtractor` is used to "uniquely associate a request sent from an IPF Connector with an asynchronously received response."
• **Message Logging**: An optional facility that "will publish both sent and received messages" for monitoring, auditing, or data warehousing.

### 3.3. Application Builder (Flo Starter)

IPF Flo Starter simplifies the bootstrapping of IPF applications.
• **Akka ActorSystem and Spring Boot**: Provides "an Akka ActorSystem and the Spring Boot autoconfiguration to wire all the necessary core dependencies."
• **Read and Write Sides**: Supports running both command (write) and query (read) sides of an application.
• **Event Processors**: Events are partitioned and processed in parallel, with configurable parallelism.
• **Remembering Entities and Passivation**: Mechanisms to load in-flight flows back into memory after a node restart. Options include "eventsourced" (default, more performant) or "ddata."

- **Flow Scheduling**: The `ipf-flo-scheduler` module offers:
  - **Action Timeouts**: Allows flow to continue if a response from an action is not provided in a defined timeframe.
  - **Retries**: Mechanism for invoking retries of actions.
  - **Time-based Scheduling**: Trigger flows based on specific times or intervals using Quartz scheduler with cron expressions and calendars.
  - **Processing Time**: Configurable durations for transitions between states, allowing for "processing time elapsed event" if exceeded.
- **Context-Based Flow Version Selection**: Enables "configuration-driven routing rules that rely on message headers to perform the routing to different versions of a flow." This is crucial for rolling upgrades.
- **Automated Retries**: Recovery of client implementation through "Action retries" and "Action revivals."
- **Monitoring and Observability**: Exposes metrics (default to Cinnamon/Prometheus), provides logging via Logback, and integrates with Spring Boot Actuator for health indicators (liveness and readiness).
- **Transaction Caching**: Provides a simple caching interface with Caffeine or InfiniSpan backends for performance optimization. InfiniSpan supports clustering and persistence to file storage.
- **IPF Bulker**: "responsible for bringing together individual transactions or components" into structured files, acting as temporary storage before streaming items to a file. Supports adding, updating, removing, closing, opening, finalizing, terminating, rejecting, and archiving bulks.
- **IPF Debulker**: Provides "the ability to process large files which contains multiple messages and transactions... breaking it into individual components." It supports various splitter implementations (JSON, XML) and client processing notifications.
- **IPF Persistent Scheduler**: Based on Quartz, it allows scheduling of one-time or recurrent jobs with persistence to a database for recovery after application crashes or restarts.

## 4. Supporting Modules and Features

- **ISO 20022 Message Model**: IPF generates its Java representation of the ISO 20022 message model directly from the E-repository. It includes validation capabilities (Schema Rules, Message Rules, Business Rules) and serialization/deserialization for JSON and XML formats. Supported message definitions include various pacs, pain, and camt messages.
- **Mapping Framework**: A configuration-driven framework using Freemarker and Orika for "many-to-many translation of business data elements" between Java classes. Supports conditional mappings and enrichments.
- **Bank Filtering**: Provides dynamic, configurable rules to filter transactions based on criteria like country, currency, BIC, NCC, and direction (debtor, creditor, or both). It integrates with Dynamic Process Settings.
- **Human Task Manager (HTM)**: "allows clients to create and manage tasks that require human intervention using a single centralized system." HTM supports task creation, allocation, approval, rejection, and purging.
- **Operational Dashboard**: "the reference implementation used by Icon to demonstrate functionality available within the Icon GUI Framework." It provides screens for cluster health, metrics, audit logs, version information, task management, and ODS search, offering insights into payment processing.
- **Operational Data Store (ODS)**: IPF’s optional offering for storing operational data, "designed for an operator or processing system to retrieve data related to processing in near real time." It distinguishes between **Message Data Structure (MDS)** objects (from ISO 20022, canonical records) and **Process Objects** (related to payment processing).

• **Payment Releaser**: Manages the release of stored payment instructions and transactions (e.g., batch transaction release), working with payment data sources to aggregate information.
• **Identifiers in IPF**: A comprehensive catalog of identifiers, including:
  ◦ **Processing Context**: `unitOfWorkId`, `associationId`, `clientRequestId`.
  ◦ **Process
**Flows**: `persistenceId`, `entityId`, `eventId`, `flowDefinitionId`, `flowHash`, `inputHash`.
  ◦ **Application Identifiers**: `correlationId`. These identifiers are crucial for tracking and tracing payments across distributed systems.

## 5. Deployment and Operations

• **Project Scaffolder**: A Maven plugin to "bootstrap a brand-new project" with a modular structure (domain, app modules).
• **Deployment Environment**: Requires a suitable database (MongoDB supported) and container hosting (Kubernetes is a common deployment target).
• **Rolling Upgrades**: IPF and Flo-lang "come with basic support for rolling upgrades," leveraging Akka Cluster's node roles to ensure flows are started only on compatible nodes and in-flight transactions are handled.
• **Configuration**: Heavily configuration-driven, primarily using HOCON files. Spring Boot properties can also be used.
• **Health Checks**: Spring Actuator endpoints (`/actuator/health/liveness`, `/actuator/health/readiness`) provide application health status. Akka HTTP Management API verifies cluster state.
• **Monitoring**: Prometheus and Grafana integration is supported for collecting and visualizing metrics.
• **Logging**: Uses Logback for flexible logging configuration, with support for stdout, file appenders, and external systems like Splunk.
• **Akka Lease MongoDB**: Used for discovery and bootstrapping an Akka cluster in MongoDB environments, supporting multi-DC active-passive deployments.

## 6. Development Workflow and Tools

• **MPS (Meta Programming System)**: Used for developing IPF's DSL. Developers can "open the project in MPS by selecting File/Open and then navigating to <generated_archetype_root>/domain-root/mps."
• **Intentions**: MPS offers "shortcut functions" (e.g., Alt+Enter) to provide quick access to common utilities and flow validation.
• **Test Framework (`ipf-test-fw-whitebox`)**: Provides a BDD-style testing framework with pre-built steps, utilities for stubbing external services (HTTP, Kafka, JMS), and the ability to interrogate flow aggregates and system events.
• **Simulator**: IPF provides simulators for external systems (e.g., Sanctions Simulator, Fraud Simulator) to facilitate testing and development.

## 7. Conclusion

IPF provides a comprehensive, robust, and scalable platform for building modern payment solutions. Its event-sourced, Akka-based architecture, combined with a powerful DSL and extensive integration capabilities, enables rapid development, high performance, and operational resilience, addressing common challenges in distributed payment processing environments.