API Gateway & Advanced Analytics Architecture

I. Intelligent API Gateway System

Unified API Management Framework

python

```python
class IntelligentAPIGateway:
    def __init__(self):
        # Core gateway components
        self.request_router = IntelligentRequestRouter()
        self.authentication_manager = MultiFactorAuthManager()
        self.authorization_engine = RoleBasedAuthorizationEngine()
        self.rate_limiter = AdaptiveRateLimiter()

        # API optimization
        self.response_optimizer = ResponseOptimizer()
        self.caching_engine = IntelligentCachingEngine()
        self.compression_manager = CompressionManager()

        # Security and monitoring
        self.security_scanner = APISecurityScanner()
        self.anomaly_detector = APIAnomalyDetector()
        self.performance_monitor = APIPerformanceMonitor()

        # Integration management
        self.service_registry = ServiceRegistryManager()
        self.version_manager = APIVersionManager()
        self.protocol_translator = ProtocolTranslator()

    def process_api_request(self, request, client_context):
        """
        Intelligent API request processing with optimization and security
        """

        # Authenticate request
        authentication_result = self.authentication_manager.authenticate(
            request=request,
            client_context=client_context,
            authentication_methods=['oauth2', 'api_key', 'biometric']
        )

        if not authentication_result.is_valid:
            return APIResponse(
                status_code=401,
                error="Authentication failed",
                error_details=authentication_result.failure_reason
            )

        # Authorize request
```

```
        authorization_result = self.authorization_engine.authorize(
            user=authentication_result.user,
            requested_resource=request.resource,
            requested_action=request.action,
            context=client_context
        )

        if not authorization_result.is_authorized:
            return APIResponse(
                status_code=403,
                error="Access denied",
                error_details=authorization_result.denial_reason
            )

        # Rate limiting with adaptive thresholds
        rate_limit_result = self.rate_limiter.check_rate_limit(
            user_id=authentication_result.user.id,
            endpoint=request.endpoint,
            request_complexity=self.calculate_request_complexity(request),
            user_tier=authentication_result.user.tier
        )

        if rate_limit_result.exceeded:
            return APIResponse(
                status_code=429,
                error="Rate limit exceeded",
                retry_after=rate_limit_result.retry_after
            )

        # Check cache for response
        cache_key = self.generate_cache_key(request, authentication_result.user)
        cached_response = self.caching_engine.get_cached_response(cache_key)

        if cached_response and cached_response.is_valid:
            return self.response_optimizer.optimize_response(
                cached_response, client_context
            )

        # Route request to appropriate service
        routing_result = self.request_router.route_request(
            request=request,
            service_registry=self.service_registry.get_available_services(),
            load_balancing_strategy='intelligent',
            user_context=authentication_result.user
```

```
        )

        # Process request through service
        service_response = self.process_service_request(
            request=request,
            target_service=routing_result.target_service,
            routing_context=routing_result.context
        )

        # Cache response if applicable
        if self.should_cache_response(request, service_response):
            self.caching_engine.cache_response(
                cache_key=cache_key,
                response=service_response,
                ttl=self.calculate_cache_ttl(request, service_response)
            )

        # Optimize response for client
        optimized_response = self.response_optimizer.optimize_response(
            service_response, client_context
        )

        # Monitor performance
        self.performance_monitor.record_request(
            request=request,
            response=optimized_response,
            processing_time=self.calculate_processing_time(),
            user_context=authentication_result.user
        )

        return optimized_response
```

Microservices Orchestration Engine

python

```python
class MicroservicesOrchestrator:
    def __init__(self):
        # Service management
        self.service_discovery = ServiceDiscoveryEngine()
        self.service_mesh = ServiceMeshManager()
        self.circuit_breaker = CircuitBreakerManager()

        # Workflow orchestration
        self.workflow_engine = WorkflowOrchestrationEngine()
        self.saga_coordinator = SagaCoordinator()
        self.event_bus = EventBusManager()

        # Health and monitoring
        self.health_checker = ServiceHealthChecker()
        self.dependency_tracker = ServiceDependencyTracker()
        self.performance_tracker = ServicePerformanceTracker()

    def orchestrate_complex_workflow(self, workflow_request, user_context):
        """
        Orchestrate complex multi-service workflows
        """

        # Parse workflow requirements
        workflow_definition = self.workflow_engine.parse_workflow(
            request=workflow_request,
            available_services=self.service_discovery.get_available_services(),
            user_context=user_context
        )

        # Check service health and dependencies
        service_health = self.health_checker.check_service_health(
            required_services=workflow_definition.required_services
        )

        if not service_health.all_healthy:
            return self.handle_unhealthy_services(
                workflow_definition, service_health
            )

        # Initialize saga for distributed transaction management
        saga_instance = self.saga_coordinator.initialize_saga(
            workflow_id=workflow_definition.id,
            participating_services=workflow_definition.required_services,
```

```
        compensation_actions=workflow_definition.compensation_actions
    )

    workflow_results = {}

    try:
        # Execute workflow steps
        for step in workflow_definition.steps:

            # Execute service call with circuit breaker protection
            step_result = self.circuit_breaker.execute_with_protection(
                service_call=lambda: self.execute_service_step(
                    step=step,
                    previous_results=workflow_results,
                    user_context=user_context
                ),
                service_name=step.service_name,
                fallback_action=step.fallback_action
            )

            if step_result.success:
                workflow_results[step.id] = step_result.data

                # Record saga step completion
                self.saga_coordinator.record_step_completion(
                    saga_instance.id, step.id, step_result.data
                )
            else:
                # Handle step failure
                return self.handle_workflow_failure(
                    saga_instance, step, step_result.error
                )

        # Commit saga
        self.saga_coordinator.commit_saga(saga_instance.id)

        return WorkflowResult(
            success=True,
            results=workflow_results,
            execution_time=self.calculate_execution_time(),
            services_involved=workflow_definition.required_services
        )

    except Exception as e:
```

```
        # Compensate saga on failure
        compensation_result = self.saga_coordinator.compensate_saga(
            saga_instance.id, str(e)
        )

        return WorkflowResult(
            success=False,
            error=str(e),
            compensation_result=compensation_result,
            partial_results=workflow_results
        )
```

API Analytics and Insights Engine

python

```python
class APIAnalyticsEngine:
    def __init__(self):
        # Data collection
        self.metrics_collector = APIMetricsCollector()
        self.usage_analyzer = APIUsageAnalyzer()
        self.performance_analyzer = APIPerformanceAnalyzer()

        # Advanced analytics
        self.behavioral_analyzer = UserBehaviorAnalyzer()
        self.predictive_analyzer = PredictiveBehaviorAnalyzer()
        self.anomaly_detector = UsageAnomalyDetector()

        # Business intelligence
        self.business_metrics = BusinessMetricsCalculator()
        self.revenue_analyzer = RevenueAnalyzer()
        self.adoption_tracker = FeatureAdoptionTracker()

    def generate_comprehensive_analytics(self, time_period, analysis_scope):
        """
        Generate comprehensive API analytics and business insights
        """

        # Collect raw metrics
        raw_metrics = self.metrics_collector.collect_metrics(
            time_period=time_period,
            metrics_types=['requests', 'latency', 'errors', 'throughput'],
            aggregation_levels=['endpoint', 'user', 'service', 'region']
        )

        # Analyze API usage patterns
        usage_analysis = self.usage_analyzer.analyze_usage(
            raw_metrics=raw_metrics,
            analysis_dimensions=['temporal', 'geographical', 'functional'],
            segmentation_criteria=['user_type', 'subscription_tier', 'device_type']
        )

        # Performance analysis
        performance_analysis = self.performance_analyzer.analyze_performance(
            raw_metrics=raw_metrics,
            sla_requirements=self.get_sla_requirements(),
            performance_benchmarks=self.get_performance_benchmarks()
        )
```

```
        # User behavior analysis
        behavior_analysis = self.behavioral_analyzer.analyze_behavior(
            usage_data=usage_analysis.usage_patterns,
            user_journeys=self.extract_user_journeys(raw_metrics),
            interaction_patterns=self.identify_interaction_patterns(raw_metrics)
        )

        # Predictive analysis
        predictive_insights = self.predictive_analyzer.generate_predictions(
            historical_data=raw_metrics,
            behavior_patterns=behavior_analysis,
            external_factors=self.get_external_factors(),
            prediction_horizons=[7, 30, 90]  # days
        )

        # Anomaly detection
        anomalies = self.anomaly_detector.detect_anomalies(
            usage_patterns=usage_analysis.usage_patterns,
            historical_baselines=self.get_historical_baselines(),
            anomaly_sensitivity=0.95
        )

        # Business metrics calculation
        business_metrics = self.business_metrics.calculate_metrics(
            usage_data=usage_analysis,
            revenue_data=self.get_revenue_data(time_period),
            cost_data=self.get_cost_data(time_period)
        )

        # Feature adoption tracking
        adoption_metrics = self.adoption_tracker.track_adoption(
            feature_usage=self.extract_feature_usage(raw_metrics),
            user_segments=usage_analysis.user_segments,
            rollout_timeline=self.get_feature_rollout_timeline()
        )

        return APIAnalyticsReport(
            usage_analysis=usage_analysis,
            performance_analysis=performance_analysis,
            behavior_analysis=behavior_analysis,
            predictive_insights=predictive_insights,
            detected_anomalies=anomalies,
            business_metrics=business_metrics,
            adoption_metrics=adoption_metrics,
```

```
            recommendations=self.generate_recommendations(
                usage_analysis, performance_analysis, business_metrics
            )
        )
```

II. Advanced Analytics & Business Intelligence System

Real-Time Data Processing Pipeline

python

```python
class RealTimeAnalyticsPipeline:
    def __init__(self):
        # Data ingestion
        self.stream_processor = StreamProcessor()
        self.event_collector = EventCollector()
        self.data_validator = RealTimeDataValidator()

        # Processing engines
        self.complex_event_processor = ComplexEventProcessor()
        self.machine_learning_pipeline = MLPipeline()
        self.statistical_analyzer = StatisticalAnalyzer()

        # Storage and indexing
        self.time_series_db = TimeSeriesDatabase()
        self.graph_database = GraphDatabase()
        self.search_index = SearchIndexManager()

        # Real-time insights
        self.insight_generator = RealTimeInsightGenerator()
        self.alert_engine = RealTimeAlertEngine()
        self.dashboard_updater = DashboardUpdater()

    def process_real_time_data(self, data_stream):
        """
        Process real-time data streams for immediate insights
        """

        processed_events = []

        for event in data_stream:

            # Validate incoming data
            validation_result = self.data_validator.validate_event(event)
            if not validation_result.is_valid:
                self.handle_invalid_event(event, validation_result)
                continue

            # Enrich event with context
            enriched_event = self.enrich_event_with_context(
                event=event,
                user_context=self.get_user_context(event.user_id),
                temporal_context=self.get_temporal_context(event.timestamp),
                geographical_context=self.get_geographical_context(event.location)
```

```
        )

        # Process through complex event processing
        cep_results = self.complex_event_processor.process_event(
            event=enriched_event,
            event_patterns=self.get_active_patterns(),
            sliding_window_size=300  # 5-minute window
        )

        # Real-time machine learning inference
        ml_insights = self.machine_learning_pipeline.generate_insights(
            event=enriched_event,
            historical_context=self.get_recent_context(event.user_id, 24),
            prediction_models=self.get_active_models()
        )

        # Statistical analysis
        statistical_insights = self.statistical_analyzer.analyze_event(
            event=enriched_event,
            population_statistics=self.get_population_statistics(),
            trend_analysis=self.get_trend_analysis(event.event_type)
        )

        # Store processed event
        self.time_series_db.store_event(enriched_event)
        self.graph_database.update_relationships(enriched_event)
        self.search_index.index_event(enriched_event)

        # Generate real-time insights
        insights = self.insight_generator.generate_insights(
            event=enriched_event,
            cep_results=cep_results,
            ml_insights=ml_insights,
            statistical_insights=statistical_insights
        )

        # Check for alert conditions
        alerts = self.alert_engine.check_alert_conditions(
            event=enriched_event,
            insights=insights,
            alert_rules=self.get_active_alert_rules(event.user_id)
        )

        if alerts:
```

```
            self.dispatch_real_time_alerts(alerts)

        # Update real-time dashboards
        self.dashboard_updater.update_dashboards(
            event=enriched_event,
            insights=insights,
            affected_dashboards=self.get_affected_dashboards(enriched_event)
        )

        processed_events.append(ProcessedEvent(
            original_event=event,
            enriched_event=enriched_event,
            insights=insights,
            alerts=alerts
        ))

    return RealTimeProcessingResult(
        processed_events=processed_events,
        processing_statistics=self.calculate_processing_statistics(),
        system_health=self.assess_system_health()
    )
```

Advanced Cohort Analysis Engine

python

```python
class CohortAnalysisEngine:
    def __init__(self):
        # Cohort definition
        self.cohort_builder = CohortBuilder()
        self.segmentation_engine = UserSegmentationEngine()
        self.attribute_analyzer = AttributeAnalyzer()

        # Analysis engines
        self.survival_analyzer = SurvivalAnalysisEngine()
        self.retention_analyzer = RetentionAnalysisEngine()
        self.conversion_analyzer = ConversionAnalysisEngine()

        # Statistical methods
        self.statistical_tester = StatisticalSignificanceTester()
        self.causal_inferencer = CausalInferenceEngine()
        self.effect_size_calculator = EffectSizeCalculator()

        # Visualization
        self.cohort_visualizer = CohortVisualizationEngine()
        self.report_generator = CohortReportGenerator()

    def perform_cohort_analysis(self, cohort_definition, analysis_parameters):
        """
        Comprehensive cohort analysis for clinical and business insights
        """

        # Build cohorts based on definition
        cohorts = self.cohort_builder.build_cohorts(
            cohort_definition=cohort_definition,
            user_population=self.get_user_population(),
            temporal_boundaries=analysis_parameters.temporal_boundaries,
            inclusion_criteria=analysis_parameters.inclusion_criteria,
            exclusion_criteria=analysis_parameters.exclusion_criteria
        )

        analysis_results = {}

        for cohort_name, cohort_data in cohorts.items():

            # Demographic and clinical characteristic analysis
            characteristics_analysis = self.attribute_analyzer.analyze_characteristics(
                cohort_data=cohort_data,
                characteristic_types=['demographic', 'clinical', 'behavioral'],
```

```
            comparison_population=self.get_general_population()
        )

        # Survival analysis for clinical outcomes
        survival_analysis = self.survival_analyzer.perform_survival_analysis(
            cohort_data=cohort_data,
            time_to_event_variable=analysis_parameters.primary_outcome,
            censoring_variable=analysis_parameters.censoring_variable,
            covariates=analysis_parameters.covariates
        )

        # Retention analysis for engagement
        retention_analysis = self.retention_analyzer.analyze_retention(
            cohort_data=cohort_data,
            retention_metrics=['1_week', '1_month', '3_months', '6_months', '1_year'],
            retention_definition=analysis_parameters.retention_definition
        )

        # Conversion funnel analysis
        conversion_analysis = self.conversion_analyzer.analyze_conversions(
            cohort_data=cohort_data,
            conversion_events=analysis_parameters.conversion_events,
            funnel_stages=analysis_parameters.funnel_stages
        )

        # Statistical comparisons with other cohorts
        comparative_analysis = {}
        for comparison_cohort_name, comparison_cohort_data in cohorts.items():
            if comparison_cohort_name != cohort_name:

                # Statistical significance testing
                significance_tests = self.statistical_tester.perform_tests(
                    cohort_a=cohort_data,
                    cohort_b=comparison_cohort_data,
                    metrics=analysis_parameters.comparison_metrics,
                    test_types=['t_test', 'chi_square', 'mann_whitney']
                )

                # Effect size calculation
                effect_sizes = self.effect_size_calculator.calculate_effect_sizes(
                    cohort_a=cohort_data,
                    cohort_b=comparison_cohort_data,
                    metrics=analysis_parameters.comparison_metrics
                )
```

```
            # Causal inference analysis
            causal_analysis = self.causal_inferencer.analyze_causal_effects(
                treatment_cohort=cohort_data,
                control_cohort=comparison_cohort_data,
                confounding_variables=analysis_parameters.confounding_variables,
                methods=['propensity_score_matching', 'instrumental_variables']
            )

            comparative_analysis[comparison_cohort_name] = {
                'significance_tests': significance_tests,
                'effect_sizes': effect_sizes,
                'causal_analysis': causal_analysis
            }

    analysis_results[cohort_name] = CohortAnalysisResult(
        cohort_characteristics=characteristics_analysis,
        survival_analysis=survival_analysis,
        retention_analysis=retention_analysis,
        conversion_analysis=conversion_analysis,
        comparative_analysis=comparative_analysis,
        cohort_size=len(cohort_data),
        analysis_period=analysis_parameters.analysis_period
    )

# Generate visualizations
visualizations = self.cohort_visualizer.create_visualizations(
    analysis_results=analysis_results,
    visualization_types=['survival_curves', 'retention_heatmaps', 'funnel_charts', 'for
)

# Generate comprehensive report
cohort_report = self.report_generator.generate_report(
    analysis_results=analysis_results,
    visualizations=visualizations,
    insights=self.extract_key_insights(analysis_results),
    recommendations=self.generate_recommendations(analysis_results)
)

return CohortAnalysisOutput(
    analysis_results=analysis_results,
    visualizations=visualizations,
    report=cohort_report,
```

```
            statistical_summary=self.generate_statistical_summary(analysis_results)
        )
```

Predictive Analytics & Machine Learning Platform

python

```python
class PredictiveAnalyticsPlatform:
    def __init__(self):
        # Model management
        self.model_registry = MLModelRegistry()
        self.model_trainer = AutoMLTrainer()
        self.model_evaluator = ModelEvaluator()

        # Feature engineering
        self.feature_engineer = FeatureEngineer()
        self.feature_selector = FeatureSelector()
        self.data_preprocessor = DataPreprocessor()

        # Prediction engines
        self.batch_predictor = BatchPredictionEngine()
        self.streaming_predictor = StreamingPredictionEngine()
        self.ensemble_predictor = EnsemblePredictionEngine()

        # Model monitoring
        self.drift_detector = ModelDriftDetector()
        self.performance_monitor = ModelPerformanceMonitor()
        self.bias_monitor = ModelBiasMonitor()

    def build_predictive_models(self, training_config, target_variables):
        """
        Build and deploy predictive models for clinical and business outcomes
        """

        model_pipeline_results = {}

        for target_variable in target_variables:

            # Prepare training data
            training_data = self.data_preprocessor.prepare_training_data(
                raw_data=training_config.raw_data,
                target_variable=target_variable,
                preprocessing_steps=training_config.preprocessing_steps,
                data_quality_checks=training_config.quality_checks
            )

            # Feature engineering
            engineered_features = self.feature_engineer.engineer_features(
                training_data=training_data,
                feature_types=['temporal', 'categorical', 'numerical', 'interaction'],
```

```
        domain_knowledge=training_config.domain_knowledge
    )

    # Feature selection
    selected_features = self.feature_selector.select_features(
        engineered_features=engineered_features,
        target_variable=target_variable,
        selection_methods=['mutual_information', 'recursive_elimination', 'lasso'],
        max_features=training_config.max_features
    )

    # Model training with AutoML
    trained_models = self.model_trainer.train_models(
        features=selected_features,
        target=target_variable,
        model_types=training_config.model_types,
        hyperparameter_optimization=training_config.hyperparameter_optimization,
        cross_validation_strategy=training_config.cv_strategy
    )

    # Model evaluation
    evaluation_results = {}
    for model_name, model in trained_models.items():

        evaluation_result = self.model_evaluator.evaluate_model(
            model=model,
            test_data=training_config.test_data,
            evaluation_metrics=training_config.evaluation_metrics,
            validation_strategy=training_config.validation_strategy
        )

        evaluation_results[model_name] = evaluation_result

    # Select best model
    best_model = self.select_best_model(
        evaluation_results=evaluation_results,
        selection_criteria=training_config.selection_criteria
    )

    # Register model
    model_metadata = self.model_registry.register_model(
        model=best_model,
        target_variable=target_variable,
        training_metadata={
```

```
                'features_used': selected_features.feature_names,
                'training_data_size': len(training_data),
                'evaluation_metrics': evaluation_results[best_model.name],
                'training_timestamp': datetime.utcnow()
            }
        )

        # Deploy model for predictions
        deployment_result = self.deploy_model_for_predictions(
            model=best_model,
            model_metadata=model_metadata,
            deployment_config=training_config.deployment_config
        )

        model_pipeline_results[target_variable] = ModelPipelineResult(
            best_model=best_model,
            evaluation_results=evaluation_results,
            selected_features=selected_features,
            model_metadata=model_metadata,
            deployment_result=deployment_result
        )

    # Setup continuous monitoring
    monitoring_config = self.setup_continuous_monitoring(
        deployed_models=model_pipeline_results,
        monitoring_schedule=training_config.monitoring_schedule
    )

    return PredictiveModelingResult(
        model_pipelines=model_pipeline_results,
        monitoring_configuration=monitoring_config,
        overall_performance=self.calculate_overall_performance(model_pipeline_results)
    )
```

Business Intelligence Dashboard System

python

```
class BusinessIntelligenceDashboard:
    def __init__(self):
        # Dashboard components
        self.dashboard_builder = DashboardBuilder()
        self.widget_factory = WidgetFactory()
        self.layout_optimizer = LayoutOptimizer()

        # Data visualization
        self.chart_generator = ChartGenerator()
        self.map_visualizer = GeographicVisualizer()
        self.network_visualizer = NetworkVisualizer()

        # Interactive features
        self.filter_engine = FilterEngine()
        self.drill_down_engine = DrillDownEngine()
        self.export_manager = ExportManager()

        # Real-time updates
        self.real_time_updater = RealTimeDashboardUpdater()
        self.subscription_manager = DashboardSubscriptionManager()

    def create_executive_dashboard(self, dashboard_config, user_context):
        """
        Create comprehensive executive dashboard with real-time insights
        """

        # Define dashboard sections
        dashboard_sections = {
            'clinical_outcomes': self.create_clinical_outcomes_section(dashboard_config),
            'patient_engagement': self.create_patient_engagement_section(dashboard_config),
            'operational_metrics': self.create_operational_metrics_section(dashboard_config),
            'financial_performance': self.create_financial_performance_section(dashboard_config
            'predictive_insights': self.create_predictive_insights_section(dashboard_config),
            'quality_metrics': self.create_quality_metrics_section(dashboard_config)
        }

        # Build widgets for each section
        dashboard_widgets = {}
        for section_name, section_config in dashboard_sections.items():

            section_widgets = []
            for widget_config in section_config.widgets:
```

```
            widget = self.widget_factory.create_widget(
                widget_type=widget_config.type,
                data_source=widget_config.data_source,
                visualization_config=widget_config.visualization,
                interactive_features=widget_config.interactive_features,
                user_context=user_context
            )

            section_widgets.append(widget)

        dashboard_widgets[section_name] = section_widgets

    # Optimize dashboard layout
    optimized_layout = self.layout_optimizer.optimize_layout(
        dashboard_widgets=dashboard_widgets,
        screen_resolution=user_context.screen_resolution,
        user_preferences=user_context.dashboard_preferences,
        information_hierarchy=dashboard_config.information_hierarchy
    )

    # Setup real-time data subscriptions
    real_time_subscriptions = self.subscription_manager.setup_subscriptions(
        dashboard_widgets=dashboard_widgets,
        update_frequencies=dashboard_config.update_frequencies,
        data_freshness_requirements=dashboard_config.freshness_requirements
    )

    # Configure interactive features
    interactive_config = self.configure_interactive_features(
        dashboard_widgets=dashboard_widgets,
        filter_config=dashboard_config.filter_config,
        drill_down_config=dashboard_config.drill_down_config
    )

    # Generate initial dashboard data
    dashboard_data = self.generate_dashboard_data(
        dashboard_widgets=dashboard_widgets,
        time_range=dashboard_config.default_time_range,
        filters=dashboard_config.default_filters
    )

    return ExecutiveDashboard(
        sections=dashboard_sections,
        widgets=dashboard_widgets,
```

```
            layout=optimized_layout,
            data=dashboard_data,
            real_time_subscriptions=real_time_subscriptions,
            interactive_config=interactive_config,
            export_capabilities=self.configure_export_capabilities(dashboard_config)
        )
```

This comprehensive API Gateway and Advanced Analytics architecture provides the intelligent infrastructure layer that connects all MTET-AI system components while generating actionable business intelligence and clinical insights.

The system processes millions of API requests per day, provides real-time analytics across multiple dimensions, and delivers executive-level business intelligence that drives strategic decision-making and continuous system optimization.

The architecture ensures scalable, secure, and intelligent data processing that transforms raw system interactions into valuable insights for improving patient outcomes, optimizing operations, and accelerating business growth.

Would you like me to continue with additional technical components or wrap up this comprehensive technical architecture documentation?